**Universidade do Minho**
Escola de Engenharia

Rui Miguel Oliveira Pinheiro

# Modelos de Comportamento para Simulação de Mobilidade em Ambientes Urbanos

Outubro de 2012

Modelos de Comportamento para Simulação de Mobilidade em Ambientes Urbanos
Rui Miguel Oliveira Pinheiro

UMinho | 2012

**Universidade do Minho**
Escola de Engenharia

Rui Miguel Oliveira Pinheiro

# Modelos de Comportamento para Simulação de Mobilidade em Ambientes Urbanos

Dissertação de Mestrado
Ciclo de Estudos Integrados Conducentes ao
Grau de Mestre em Engenharia de Comunicações

Trabalho realizado sob a orientação do
**Professor Doutor Adriano Jorge Cardoso Moreira**
e da
**Professora Doutora Maria João Mesquita Rodrigues da Cunha Nicolau**

Outubro de 2012

*“Em condições normais somos muito melhores e em condições normais vamos ser campeões... Em condições anormais também vamos ser campeões.”*

*José Mourinho*

# Agradecimentos

Em primeiro lugar, gostaria de agradecer aos meus orientadores Professor Doutor Adriano Moreira e à Professora Doutora Maria João Nicolau por toda a disponibilidade, orientação e pela dedicação no decorrer deste projeto.

Ao meu pai e ao meu irmão, um agradecimento por todo o apoio e carinho que me deram. Um agradecimento muito especial à minha mãe por todo carinho e sacrifícios que fez e que me proporcionaram tudo o que precisava para chegar até aqui.

À Daniela Lobo por ser uma companheira excepcional e extraordinária, por ter sido a minha pedra basilar, por estar sempre ao meu lado, por me animar nos momentos mais difíceis e por não me deixar tomar o percurso mais fácil. Gostaria também de agradecer à Genoveva pela sua ajuda em tornar esta dissertação ainda melhor.

A todos os meus amigos que estiveram comigo durante o meu percurso académico, nomeadamente o Carlos Samy, o Filipe Magalhães, o João Fernandes e o Rúben Barbosa, que estiveram sempre lá para mim, sem eles a universidade não teria sido tão memorável e uma experiência tão enriquecedora.

Gostaria também de agradecer ao Francisco Silva e Laurent Miranda pela ajuda e por terem sido os parceiros ideias na realização desde projeto.

# Resumo

Com a crescente utilização dos dispositivos móveis, é cada vez mais importante o estudo do desempenho dos sistemas de telecomunicações móveis, nomeadamente em ambientes urbanos. Este estudo deve recorrer a modelos de mobilidade específicos (para pedestres e veículos automóveis, por exemplo), em vez de modelos genéricos de mobilidade. No entanto, os simuladores de mobilidade existentes continuam a ser muito genéricos e incapazes de simular, em larga escala, cenários urbanos.

Partindo da premissa de que os dispositivos móveis já estão profundamente inseridos nos centros urbanos, faz sentido a criação de um simulador que recrie o movimento das entidades móveis (pedestres ou veículos), nestes cenários.

Nos ambientes urbanos existem vários tipos de condicionantes que não permitem que os veículos e pedestres se desloquem livremente, por exemplo, semáforos, sentidos obrigatórios das ruas, o facto dos veículos apenas poderem circular nas estradas, entre outros. Por outro lado, os passeios têm características próprias que influenciam o comportamento dos pedestres, por exemplo, paragens de autocarros, lojas, aglomerados de pessoas, etc..

Esta dissertação descreve o desenvolvimento de um simulador de mobilidade em ambientes urbanos que permita simular movimentos de vários tipos de atores, bem como diversas intereações entre os diversos intervenientes. Pretende-se que estas simulações sejam o mais realistas possível, ou seja, cada tipo de actor (pedestre, carro, autocarro, etc.) vai ter uma mobilidade independente, assumindo as características dos espaços urbanos, como por exemplo, diferentes sentidos de trânsito, existência de situações que fazem com que a velocidade dos actores varie, como semáforos, passadeiras, paragens de autocarro e zonas de maior concentração de actores. O simulador deverá permitir a definição de cenários com base em mapas das estradas e a geração automática de tráfego, pedestre e automóvel. A componente do simulador que vai constituir o foco deste trabalho é a concepção e desenvolvimento dos vários tipos de actores (Pedestres, Automóveis, Trams, etc.).

# Abstract

With the increasing use of mobile devices it is even more important to study the performance of mobile telecommunications systems, particularly in urban environments. This study should appeal to specific mobility models (for pedestrians and motor vehicles, for example) instead of generic models of mobility. However, the existing mobility simulators remain very generic and unable to simulate large-scale urban scenes.

Assuming that mobile devices are already deeply embedded in urban centers, it makes sense to create a simulator that recreates the motion of mobile entities (pedestrians or vehicles), in these scenarios. In urban environments there are several types of constraints that do not allow vehicles and pedestrians to move freely, such as traffic lights, street directions required, and the fact that the vehicles can only move on roads, among others. On the other hand, the sidewalks have their own characteristics that influence the behavior of pedestrians, like bus stops, shops, crowds of people, among others.

This thesis describes the development of a simulator of mobility in urban environments that allows simulating movements of various types of actors, as well as several interactions between different actors. It is intended that these simulations are as realistic as possible, in other words, each type of player (pedestrian, car, bus, among others) will have an independent mobility, assuming the characteristics of urban spaces, such as different traffic directions, the existence of situations that can diverse the speed of the players, like traffic lights, crosswalks, bus stops and areas of high concentration of players. The simulator should allow the definition of scenarios based on maps of the roads and the automatic formation of traffic, pedestrian and cars. The component of the simulator that will be the focus of this work is the design and development of various types of players (Pedestrians, Cars, Trams, among others).

# Conteúdo

# Lista de Figuras

# Lista de Acrónimos

| | |
|---|---|
| **AVI** | Audio Video Interleave |
| **BartUM** | Bartolomeu_Urban Mobility Simulator |
| **CATDTN** | Connectivity, Applications, and Trials of Delay Tolerant Networking |
| **GloMoSim** | Global Mobile Information System Simulator |
| **GPS** | Global Positioning System |
| **IP** | Internet Protocol |
| **LAN** | Local Area Network |
| **MiXim** | Mixed Simulator |
| **NS-2** | Network Simulator-2 |
| **ONE** | Opportunistic Network Environment Simulator |
| **SINDTN** | Security Infrastructure for Delay Tolerant Network |
| **SUMO** | Simulation of Urban Mobility |
| **TCP** | Transmission Control Protocol |
| **UDP** | User Datagram Protocol |
| **WKT** | Well-Known Text |
| **XML** | Extensible Markup Language |

# 1 Introdução

## 1.1 Enquadramento

A Internet pode ser encarada como uma rede de computadores de dimensão mundial que permite a troca de informação em grande escala. De facto, a Internet é constituída por milhares de redes de computadores inter ligadas entre si. Dentro dessas redes existem as que se definem por não precisarem de meio físico de transmissão, as *redes sem fios*, que adquirem cada vez mais importância, quer pelo aumento do desempenho dos sistemas de telecomunicações móveis, quer pelo uso crescente de dispositivos como telemóveis, computadores portáteis, smartphones, entre outros.

Sendo esta uma área ainda em desenvolvimento, os desafios existentes motivam estudos como é o caso do desempenho dessas redes sob diferentes cenários de mobilidade, nomeadamente em ambientes urbanos. Estes estudos recorrem a *simuladores de mobilidade* ainda muito genéricos e incapazes de simular cenários urbanos em grande escala, nos quais se destaca a necessidade de recorrer a modelos de mobilidade específicos, como de pedestres e veículos automóveis, e a criação de um *simulador* capaz de recriar o movimento das entidades móveis nestes cenários.

As simulações pretendem-se o mais aproximado possível da realidade de um ambiente urbano, visto que os modelos de mobilidade clássicos não permitem reunir todos os condicionantes que limitam a deslocação livre de pedestres, como os semáforos, os sentidos obrigatórios das ruas, a circulação limitada dos veículos apenas pelas estradas, etc.. Cada ator deverá ter uma mobilidade independente, assumindo as características dos espaços urbanos, como, por exemplo, diferentes sentidos de trânsito, existência de situações que fazem com que a velocidade dos atores varie, como semáforos, passadeiras, paragens de autocarro e zonas de maior concentração de atores, etc.. No entanto, para uma simulação de um ambiente urbano, é necessário que se tenha igualmente em conta a definição de cenários com base em mapas de estradas e a geração automática de tráfego pedestre e automóvel.

O *simulador* desenvolvido durante este estudo pretende considerar as condicionantes que outros simuladores genéricos não permitem, possibilitando a utilização de um grande número de atores e a definição de um padrão de movimento diferenciado para cada um deles.

## 1.2 Objetivos

Esta dissertação tem por objectivo a criação de um simulador inserido nos pressupostos descritos no enquadramento.

Os objectivos específicos estão relacionados com os resultados que se pretendem alcançar com o objetivo principal. São eles:

- Conceber os modelos comportamentais para os vários tipos de atores, com base em modelos existentes.
- Implementar e integrar no simulador os modelos comportamentais criados.
- Testar e avaliar os comportamentos de cada tipo de ator individualmente e as interações entre os diversos tipos de atores.
- Testar e avaliar a carga dos computadores com o uso dos diferentes tipos de atores.

## 1.3 Requisitos

De seguida serão enumerados alguns requisitos que o simulador terá de cumprir:

- Uma simulação de uma cidade para parecer minimamente real, tem de conseguir simular muitas entidades (dezenas de milhares) a movimentarem-se pela cidade, cada entidade ser dotada de capacidades específicas e deve mover-se de acordo com as regras dessa cidade.
- É importante que o simulador seja capaz de modelar as entidades com um grande nível de pormenor.

- Tem de ser flexível, ser capaz de interagir com outros softwares, por exemplo, de criação de mapas, que possam dar suporte a uma melhor simulação.
- Precisa de ter um interface gráfico para os utilizadores verem a simulação a correr e não apenas ter acesso a ficheiros com os dados da simulação.
- Tem de ser capaz de correr nos principais sistemas operativos, nomeadamente o Windows, Mac OS e Linux, ou seja, de forma a ser multiplataforma para ser acessível a qualquer utilizador.
- Ser *open-source* para que a comunidade científica possa usar e ajudar a desenvolver uma ferramenta mais completa e melhor.

## 1.4 Estrutura da dissertação

Esta dissertação está dividida em 6 capítulos. No primeiro capítulo é feita uma introdução ao tema da dissertação, em que são definidos objectivos e requisitos que se pretendem atingir.

No segundo capítulo, Conceitos e Trabalhos Relacionados, é feito um estudo sobre o estado atual da simulação de redes móveis em espaços urbanos. É feita uma abordagem aos diversos tipos de simulação, os vários modelos de mobilidade existentes, individuais e de grupo, e apresentada uma visão global de alguns simuladores de movimento existentes.

É feita, no terceiro capítulo, uma descrição da arquitetura geral do simulador implementado, sendo o capítulo denominado Bartolomeu Urban Mobility Simulator. Será feita uma descrição genérica de cada bloco existente no simulador.

No quarto capítulo, Atores e Geradores, é apresentada uma explicação da relação entre geradores e atores e uma explicação detalhada do comportamento dos atores e da sua implementação.

No quinto capítulo, Testes e Análises de resultados, são feitos os testes ao simulador e, de seguida, feita uma análise aos resultados obtidos.

No sexto e último capítulo, Conclusões e Trabalho Futuro, são apresentadas as conclusões do projecto e o trabalho que pode ser feito no futuro para melhorar o simulador.

# 2 Conceitos e Trabalhos Relacionados

Neste capítulo será feito um apanhado geral do estado atual da arte e de alguns trabalhos relacionados já existentes nesta área. O capítulo foi dividido em 4 secções: a primeira parte é uma breve introdução aos simuladores, a segunda é uma descrição dos métodos de classificação dos simuladores, a terceira faz uma descrição de vários modelos de mobilidade existentes e a quarta é uma revisão de alguns simuladores que já existem nesta área.

## 2.1 Introdução

As cidades constituem um fenómeno complexo, devido a uma grande quantidade de factores urbanos, naturais e institucionais. Estes factores vão-se modificando com o avançar do tempo e desenvolvendo novos padrões de crescimento urbano, como é o caso da mobilidade da população e veículos, bem como o crescimento das comunicações sem fios.

A tecnologia computacional tem evoluído de forma a garantir a sua utilidade no estudo destes cenários reais e complexos. *Simuladores* e *modelos* têm sido utilizados para uma melhor compreensão da expansão das cidades, através da realização de simulações da dinâmica urbana e dos seus processos de crescimento, numa tentativa de recriação de um mundo real. A *simulação* constitui uma ferramenta segura e eficiente no estudo de cenários complexos permitindo, além do mais, uma redução de custos, nomeadamente no planeamento e gestão de sistemas de transporte, através da preparação de medidas de redução de consumos energético, ruído, poluição atmosférica, etc..

Além disso, através de *modelos de simulação* é possível estudar o movimento de entidades móveis com o objectivo de introduzir melhorias na comunicação entre elas, ou mesmo a redução de custos na implementação de soluções que permitam os utilizadores terem acesso aos serviços de redes móveis a partir de qualquer lugar.

Deste modo, a simulação serve como uma imitação de algo real ou de processos, permitindo a representação de características ou comportamentos específicos de um sistema abstrato.

O presente capítulo apresenta os diferentes conceitos de *simulação*, os *modelos de mobilidade* mais relevantes da atualidade, os *simuladores* de movimento e os simuladores de redes móveis em ambientes urbanos.

## 2.2 Classificação de Modelos de Simulação

Os simuladores e os modelos de simulação podem ser classificados de acordo com vários fatores. Os fatores que são tomados em conta geralmente são se o modelo é discreto ou contínuo no tempo, se o modelo é determinístico ou estocástico, os níveis de detalhe do modelo ou a metodologia usada para a implementação do modelo.

### *2.2.1 Tempo discreto e tempo contínuo*

Quase todos os modelos de simulação descrevem sistemas dinâmicos. O tempo é sempre uma variável que se tem de ter em conta. Os modelos podem ser caraterizados como discretos, contínuos ou uma combinação dos dois. Nos modelos discretos os eventos têm início num determinado instante de tempo, mas os efeitos que resultam da ocorrência desses eventos só se fazem sentir simultaneamente depois de um dado tempo (*Figura 1*).

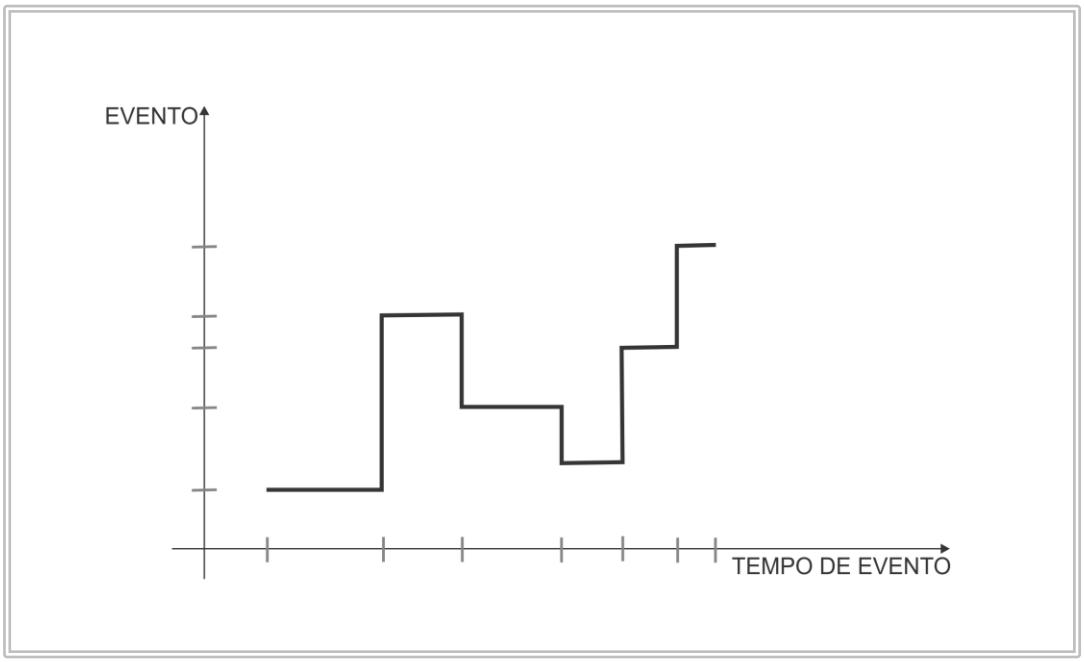


Figura 1 – Ocorrência dos eventos em modelos discretos (adaptado de Modelagem e Simulação de Sistemas).

No caso dos modelos contínuos, os efeitos da ocorrência de um evento fazem-se sentir de forma contínua ao longo do tempo (*Figura 2*).

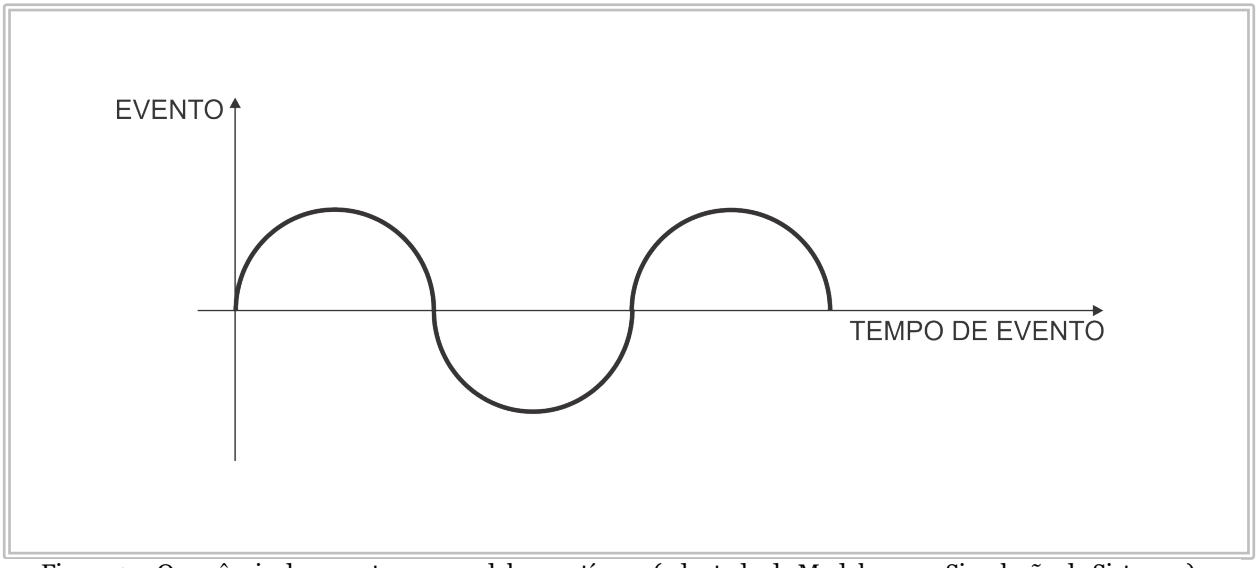


Figura 2 – Ocorrência dos eventos em modelos contínuos (adaptado de Modelagem e Simulação de Sistemas).

Como se pode ver nas *figuras 1 e 2*, a diferença entre os dois consiste em que no tempo contínuo os eventos são lineares, não existem variações bruscas no estado atual

do evento para um estado futuro, como acontece no tempo discreto. No tempo discreto, quando ocorre uma mudança de estado acontece uma alteração brusca e depois mantém o seu estado por um determinado tempo até voltar a atualizar, por exemplo um jornal pode descrever o preço do petróleo uma vez por dia ao invés da variação do seu preço durante o dia. O tempo contínuo é como acontece na vida real, mas em simulação não se conseguem reproduzir variações de tempo contínuo. Assim sendo, é usado nas simulações o tempo discreto.

### *2.2.2 Modelo determinístico e modelo estocástico*

É muito útil classificar os sistemas simulados em duas categorias separadas dependentes do grau de aleatoriedade associada ao comportamento do sistema que está a ser simulado. A simulação pode ser caracterizada como determinística ou estocástica. Os modelos determinísticos são previsíveis, não possuem variáveis com valor aleatório, ou seja, todas as ações são definidas através de expressões lógicas ou matemáticas exatas. Estes modelos são bons para simular os cenários que têm como intuito reproduzir ações previamente pensadas e sempre com os mesmos resultados, por exemplo, fazendo com que os veículos pareçam muito mecânicos e monótonos e, perdendo-se assim o realismo da simulação.

No caso dos modelos estocásticos é ao contrário, estes incluem processos que usam funções de probabilidade, ou seja, são modelos muito imprevisíveis. O estado atual da simulação não define completamente o seu próximo estado, o que faz com que duas simulações que têm exatamente os mesmos parâmetros de entrada, tenham resultados diferentes.

### *2.2.3 Níveis de simulação*

O nível de detalhe que normalmente é escolhido para representar um modelo de simulação depende essencialmente da capacidade de processamento do computador usado. Quantos mais recursos se tiver, maior poderá ser o nível de detalhe de cada elemento a ser simulado. Os modelos podem ser classificados em relação ao nível de

detalhe como macroscópicos, mesoscópicos, microscópicos e nanoscópicos (Daiheng Ni, 2006).

#### 2.2.3.1 Simulação macroscópica

Os *modelos macroscópicos* descrevem as entidades e as suas atividades com pouco nível de detalhe. São bons para simulações de grandes áreas de tráfego e para fazer o seu planeamento, no caso de não ser necessária uma modelação muito detalhada. Em vez de modelar entidades individualmente, os modelos macroscópicos usam agregação de variáveis como fluxo, velocidade e densidade para caracterizar o tráfego. Não é uma abordagem muito realista, pois na vida real existem diferentes tipos de veículos conduzidos por diferentes tipos de pessoas, com estilos e comportamentos próprios. Os modelos macroscópicos tratam todos os veículos como um todo, em alguns modelos mais avançados podem até descriminar diferentes tipos de veículos mas acaba por tratar todos de maneira igual. Estes modelos têm a vantagem de serem muito rápidos, exatos e requererem baixo esforço computacional em relação a modelos mais detalhados. No entanto, não é o mais indicado para modelos urbanos em geral, por exemplo, mudanças de faixas por parte de veículos não são representados e o modelo apenas fornece informações quantitativas e qualitativas globais.

#### 2.2.3.2 Simulação microscópica

Os *modelos microscópicos* descrevem a maior parte das entidades com alto nível de detalhe, os veículos e as suas interações são modelados individualmente. Estes modelos podem realizar simulações com maior realismo que os modelos macroscópicos e mesoscópicos, devido aos detalhes extras que são adicionados às entidades. Baseiam-se na descrição do movimento de cada veículo que faz parte do fluxo de tráfego, o que implica modelar ações, por exemplo, a aceleração e a mudança de via de cada condutor em resposta ao tráfego circundante. Os veículos e condutores possuem muitas propriedades, podendo algumas ser partilhadas entre as entidades, por exemplo, o tipo de veículo, velocidade, etc..

São usados submodelos para definir comportamentos e regras para as entidades. O *car-following* é um submodelo muito usado, que tem como objetivo gerar a resposta de um condutor em relação ao veículo que segue à sua frente. Consoante a informação obtida, o condutor acelera ou abranda de modo a obter a velocidade desejada ou evitar uma colisão com o veículo que se encontra à sua frente. Outro submodelo usado é o *lane-changing* que serve para determinar se existe a possibilidade de um condutor mudar de via. A mudança de via pode ser obrigatória caso o condutor necessite de mudar de via, ou pode simplesmente acontecer por um condutor querer ultrapassar um veículo à sua frente. Um outro submodelo é o *gap-acceptance* que serve para determinar se um dado veículo pode avançar num cruzamento ou numa interseção. A decisão é tomada de acordo com o intervalo de tempo disponível para entrar na nova via sem por em causa a sua segurança. É o mais indicado para simular ambientes urbanos, devido ao facto de existirem diversas entidades com diferentes comportamentos e interações.

A complexidade dos simuladores de movimento microscópico vem do facto de ser necessário simular um grande número de veículos e a dificuldade de manter o controlo de todos os seus estados e interações. Devido ao alto nível de detalhe e ao grande número de interações entre as entidades, é uma abordagem que exige muito esforço computacional e requer um processo de configuração complexo, difícil e demorado.

#### 2.2.3.3 Simulação mesoscópica

Os *modelos mesoscópicos* preenchem a lacuna entre os modelos macroscópicos e microscópicos. Estes modelos descrevem as entidades de tráfego com maior nível de detalhe que os modelos macroscópicos, mas os seus comportamentos e interações com outros veículos têm menor nível de detalhe que os modelos microscópicos. Existem algumas abordagens que podem ser aplicadas neste tipo de simulações. Uma das abordagens agrupa os veículos em pacotes, que serão posteriormente tratados como uma entidade única e todos os veículos que fazem parte de um pacote terão a mesma velocidade e o mesmo comportamento. Outra abordagem agrupa cada veículo em células que controlam o seu comportamento e cada célula é responsável por determinar a velocidade de cada veículo. Uma última abordagem trata a estrada como uma fila. Os

veículos são representados individualmente e mantêm a sua velocidade individual, mas o seu comportamento não é modelado com detalhe. A velocidade dos veículos que vão nessa estrada é calculada usando uma função macroscópica de relação velocidade-densidade e os veículos são transferidos de uma estrada para a outra por um controlador de filas.

Quando se quer apreender com mais detalhe alguma área específica da rede mantendo, no entanto, uma vista ampla do fluxo de tráfego, uma escala intermédia como a dos modelos mesoscópicos pode ser a mais indicada. Este modelo é adequado para fazer a análise de redes interurbanas, pois apresenta algumas limitações para simular os meios urbanos que exigem um maior nível de detalhe.

#### 2.2.3.4 Simulação nanoscópica

Os *modelos nanoscópicos* tentam trazer a simulação um pouco mais perto da realidade, imitando as experiências do mundo real. Estes modelos são diferentes dos descritos anteriormente pois o sistema é composto pelo condutor, o veículo e ambiente em redor (estrada, sinais, outros veículos, pedestres, infraestruturas, etc.). O comportamento do condutor, a dinâmica do veículo e as interações entre o condutor e o veículo são muito detalhadas. Os modelos têm a preocupação de modelar a cognição, percepção, decisão e erros do condutor. O condutor guarda informações e faz o controlo de decisões em relação à direção, à aceleração e à desaceleração. Os veículos respondem dinamicamente ao condutor, executando as decisões de controlo e movendo-se no ambiente. A informação de como está a funcionar o veículo, em conjunto com a informação do ambiente em redor, constituem as bases para o condutor tomar as decisões de controlo para o próximo passo.

Estes modelos são usados para fornecer conhecimentos detalhados sobre operações de trânsito em condições extremas, como ausência de regras de trânsito e acidentes de trânsito, e serve de auxílio para uma melhor tomada de decisões em gestão de situações de emergência. Devido ao grande nível de detalhe dos veículos e da sua dinâmica, estes modelos são muito usados por empresas de automóveis para desenvolver simuladores para teste dos próprios produtos.

#### 2.2.3.5 Síntese

Em suma, para uma percepção melhor sobre os diferentes níveis de detalhe, vai ser feita uma pequena descrição usando uma analogia.

Supõe-se que se está a observar o tráfego a uma altura de 10 000 metros do chão, como se verifica no mapa de fundo da *Figura 3*. O tráfego apresenta-se como um aglomerado homogéneo que se propaga para a frente e para trás, sem distinção das entidades. Este cenário corresponde a uma simulação macroscópica. Se baixar até aos 3000 metros (imagem no canto superior esquerdo da *Figura 3*), deixa de haver a sensação de movimento de um aglomerado e começa-se a ver pequenas partículas. Os veículos comportam-se como partículas a "saltar" de uma célula para outra, governados por uma lógica pré-determinada. Este cenário corresponde a uma simulação mesoscópica. Se baixar até aos 1000 metros (imagem no meio da *Figura 3*), o cenário é caracterizado por entidades em movimento, com interação entre elas de modo a manterem uma posição segura no fluxo de tráfego. Isto é o cenário na simulação microscópica. Descendo até ao chão (imagem no canto inferior esquerdo da *Figura 3*), deixa-se de ver as diversas entidades como um aglomerado ou como partículas a "saltar" de célula em célula, mas vê-se cada entidade com pormenor, como numa visão real de um condutor. Isto é um cenário nanoscópio.

Figura 3 - Diferentes níveis de detalhe (retirado de "A Framework for New Generation Transportation Simulation")

### 2.2.4 Tipos de simulação

O núcleo dos simuladores de tráfego refere-se ao código responsável pelo avanço do estado da simulação. Existem diferentes técnicas para implementar o núcleo do simulador. De seguida, é feito um apanhado geral das técnicas mais relevantes na área da simulação de tráfego.

#### 2.2.4.1 Simulação *time-driven*

Com esta técnica o simulador avança em intervalos de tempo fixos. Ao fim de cada um desses intervalos de tempo procura eventos que possam ter ocorrido, ou seja, cada componente da simulação é verificado no caso de existirem alterações, no fim do ciclo este atualiza os seus componentes e o tempo global da simulação. Este tipo de modelação é muito linear e simples de implementar e foi muito usado nos primeiros modelos de simulação.

Uma desvantagem é que se dois eventos ocorrem no mesmo intervalo de tempo mas em tempos diferentes, estes são apenas tratados no fim desse intervalo, o que não é real. Em alguns casos, o estado do sistema pode manter-se constante durante muito tempo e, se os intervalos entre eventos forem muito grandes, o poder computacional é gasto para percorrer esses intervalos sem fazer nada. Uma solução é ter intervalos de tempo menores. O modelo não é recomendado para casos onde os eventos ocorrem em momentos aleatórios.

#### 2.2.4.2 Simulação *event-driven*

Na simulação *event-driven* é criada uma série de eventos que são executados numa ordem cronológica. O simulador apenas atualiza o seu estado quando ocorrem eventos, ao contrário do *time-driven* que tem sempre ciclos de tempo fixos, independentemente de existirem ou não eventos. Cada evento tem um tempo de execução e, consoante esse tempo, eles são ordenados numa fila. O simulador gere os eventos, retirando os eventos da fila no momento devido e dando ordem para ser executado o evento. O tempo global de simulação avança para o tempo do evento que está a ser tratado. No caso dos eventos que devem ocorrer no mesmo momento, são executados pela ordem com que foram agendados, usando o mesmo princípio do *first-in first-out*.

Um evento pode dar origem a vários eventos. Os novos eventos são introduzidos na lista e estão prontos a serem executados após a execução de todos os outros eventos.

A simulação acaba quando não existirem mais eventos agendados para executar ou chegar ao fim o tempo de simulação.

Este método tem bons resultados em relação à gestão do tempo, mas gera muitos problemas de escalabilidade.

#### *2.2.4.3* Simulação *process-driven*

A simulação *process-driven* segue uma lógica pré-definida de eventos e executa os passos da simulação sequencialmente. A abordagem usada nos simuladores *process-driven* é primeiro definir as entidades do sistema, depois criar um processo para cada entidade e de seguida executar os processos. Os processos são executados concorrentemente durante a simulação e podem comunicar entre si por mensagens. Como acontece na simulação *event-driven*, também existe um mecanismo de escalonamento e uma estrutura em lista mas com a diferença que, de cada vez que se coloca uma entidade na lista, esta define um novo processo ou o seu ponto de reativação (caso em que o processo é bloqueado e é executado novamente). À medida que o tempo de simulação avança, as condições para início e fim de um evento são examinados.

Este tipo de simulação requer que o utilizador que modela conheça a sequência lógica da simulação com antecedência e a simulação está completa quando essa sequência for percorrida.

#### 2.2.4.4 Simulação *object-driven e agent-driven*

A simulação *object-driven,* em contraste com a simulação *process-driven*, não tem definida uma sequência lógica de execução. A simulação depende da interação de objetos, que são as entidades no sistema. Semelhante à programação orientada a objetos, um objeto possui propriedades e comportamentos por meio de métodos. Um objeto pode derivar de outro objeto herdando as suas propriedades e métodos ou até substituir essas propriedades no objeto que o originou.

Um tipo especial de *object-driven* é o *agent-driven*, que normalmente aplica um sistema de inteligência humana. O sistema é modelado como uma coleção de entidades

que tomam decisões autonomamente, chamados de agentes. Individualmente, cada agente avalia a situação e toma decisões com base num grupo de regras básicas. Os agentes usam apenas regras simples e podem apresentar resultados complexos e muito úteis. Também estes executam comportamentos apropriados para o sistema que estão a representar. O modelador apenas introduz as suposições mais relevantes para a situação com que está a lidar e depois vê o fenómeno a crescer a partir da interações dos agentes. Interações de competitividade entre agentes é uma das características da simulação *agent-driven*, que conta com o poder computacional para explorar novas dinâmicas que não estão ao alcance de métodos totalmente matemáticos.

Num nível mais simples, a simulação *agent-driven* consiste num sistema de agentes e as relações entre eles. Mesmo um modelo simples pode exibir comportamentos complexos e providenciar informações valiosas sobre a dinâmica dos sistemas do mundo real que está a simular. Os agentes têm a capacidade de evoluir, permitindo assim o aparecimento de comportamentos imprevisívies. Este modelo por vezes incorpora redes neuronais, algoritmos evolutivos ou outras técnicas que permitem uma aprendizagem realista e adaptativa.

Também tem uma grande flexibilidade, podendo ser usado em diversas áreas e para diferentes propósitos. Os agentes são orientados por objetivos e conseguem atingi-los mudando os seus comportamentos e adaptando-se ao ambiente em redor. O simulador *agent-driven* é um complemento aos métodos de análise tradicionais.

#### *2.2.4.5* Simulação *data-driven*

Nos modelos de simulação descritos anteriormente todas as atividades e entidades são controladas pelo simulador. Normalmente essas atividade ocorrem no mesmo nível de detalhe de simulação.

Uma recente abordagem tem emergido na área dos simuladores de tráfego, chamada de *data-driven*. É um modelo que pode ser distribuído e consegue capturar diferentes níveis de detalhe da simulação.

O progresso do simulador *data-driven* é controlado pelos dados de entrada da simulação. Todos os cálculos e suposições do modelo baseiam-se nos dados que o

utilizador introduz no início e durante a simulação. Diferentes dados podem desencadear diferentes componentes do modelo, diferentes níveis de detalhe e diferentes alcances de simulação, podendo todos eles coexistir num sistema de simulação único. Os dados que são fornecidos como *inputs* do sistema podem ir desde parâmetros de início e fim da simulação, regras que têm de ser cumpridas, informação estatística para permitir uma melhor previsão de um resultado futuro, entre outros.

A grande vantagem é que os utilizadores podem personalizar o modelo, bastando apenas mudar os parâmetros de entrada, sem necessitarem de refazer o código do modelo. Os utilizadores não precisam de ter grandes conhecimentos de programação, mudando apenas a estrutura de dados do modelo. Este facto permite uma maior flexibilidade e maior capacidade para simulações complexas.

## 2.3 Modelos de Mobilidade

Os modelos de mobilidade têm sido desenvolvidos para simular o mundo real, para determinar se as soluções que se pretendem implementar na vida real serão úteis e eficazes. Os modelos de mobilidade são usados para simular diferentes cenários antes de implementa-los. Muitos modelos têm sido desenvolvidos para simular o comportamento de determinadas entidades. As diferentes entidades são referidas no modelo de mobilidade como os nós móveis, que podem ser representação de carros, pessoas, dispositivos móveis (como por exemplo telemóveis, dispositivos GPS), etc..

Atualmente os modelos de mobilidades estão divididos em dois tipos, os modelos baseados em rastros (traces) e os modelos sintéticos.

Os modelos baseados em rastros dão a possibilidade de compreender padrões de movimento realistas e são a base de obtenção de dados estatísticos de movimento. Os rastros são os padrões de mobilidade que são observados na vida real. Os rastros podem fornecer informações precisas, no caso de se usar um grande número de participantes no estudo e um tempo de observação grande.

No entanto, o uso dos rastros pode ser muito limitado, pois a maioria deles é aplicável em cenários muito específicos.

No caso de se tentar modelar um novo cenário, se não existir o estudo dos rastros daquele cenário em específico, torna-se complicado fazer a modelação. Nestes casos em específico é necessário recorrer ao uso de modelos sintéticos. Os modelos sintéticos tentam representar, de forma realista, as diferentes entidades sem o uso de rastros, usando métodos analíticos para descrever os movimentos.

Alguns modelos sintéticos tentam simular o comportamento humano real com base em características de mobilidade observadas a partir dos rastros. Nesta situação pode-se considerar que são modelos de mobilidade híbridos.

Os modelos de mobilidade podem ser divididos em padrões de mobilidade individual (modelos individuais) e padrões de mobilidade em grupo (modelos de grupo).

De seguida, será feito um levantamento dos modelos de mobilidade mais relevantes, atualmente. A divisão dos diversos modelos é feita em modelos de mobilidade individual e modelos de mobilidade em grupo.

Para o estudo destes modelos foram usadas as referências (Camp, Boleng e Davies 2002), (Ribeiro e Sofia 2011), (Chiang e Gerla 1998), (Musolesi e Mascolo 2008).

### *2.3.1 Modelos de mobilidade individual*

Os modelos individuais tentam fazer uma representação real através da simulação das entidades individualmente. As suas ações são completamente independentes dos outros intervenientes da simulação, ou seja, não dependem nem afectam o comportamento das outras entidades. As principais características destes modelos residem nos nós possuirem direções e velocidades aleatórias. Nos modelos mais recentes tem-se notado uma crescente preocupação em dotar os nós com algumas caraterísticas não aleatórias, como por exemplo, uso de tempos de espera antes de alterar a direção e ter memória temporal dos movimentos anteriores, para no próximo destino ser escolhida uma direção que não obrigue o nó a fazer um movimento brusco, e

conferem uma sensação de suavidade na transição para os próximos destinos, como acontece na vida real.

#### *2.3.1.1 Random Walk Mobility Model*

O *Random Walk* foi dos primeiros modelos a ser desenvolvido. Foi pensado inicialmente por Einstein em 1926. Baseado em movimentos imprevisíveis da natureza, este modelo recria-os através da escolha aleatória de direção e velocidade dos nós aquando da deslocação para uma nova localização.

A velocidade e direção são escolhidas aleatoriamente num intervalo pré-definido nos limites [*speedmin*, *speedmax*] e [0, $2\pi$] respectivamente. O movimento originado acorre em intervalos de tempo constantes ou intervalos de distância percorrida constantes, reforçando que no fim de cada intervalo é calculada uma nova direção e velocidade. Sempre que os nós atingem os limites da simulação, são reenviados com um ângulo recalculado a partir da direção anterior, prosseguindo novamente o caminho.

É de realçar a particularidade deste modelo permitir a representação do movimento no qual as entidades se movem à volta de um ponto, garantindo o retorno do nó ao ponto de partida. No entanto, este mesmo modelo não armazena em memória os movimentos anteriores, localizações passadas e valores de velocidade, tornando-os independentes dos valores da nova localização e velocidade (Camp, Boleng e Davies 2002). Significa, com isto, que poderá originar movimentos pouco realistas, como paragens bruscas e mudanças de direção impossíveis na realidade.

Servindo de base para a maioria dos restantes modelos, este é, contudo, limitado na representação do movimento aleatório humano e individualizado (Ribeiro e Sofia 2011).

#### *2.3.1.2 Probabilistic Random Walk Mobility Model*

Este modelo é caracterizado por utilizar uma matriz de probabilidades para determinar a posição seguinte do nó, com a representação de 3 estados possíveis para a

posição do nó. O *estado 0* representa a posição atual do nó, *estado 1* representa a posição anterior e o *estado 2* a posição seguinte se o nó continuar na mesma direção.

A *Figura 4* representa um esquema operacional do funcionamento do modelo, juntamente com a matriz de probabilidades:

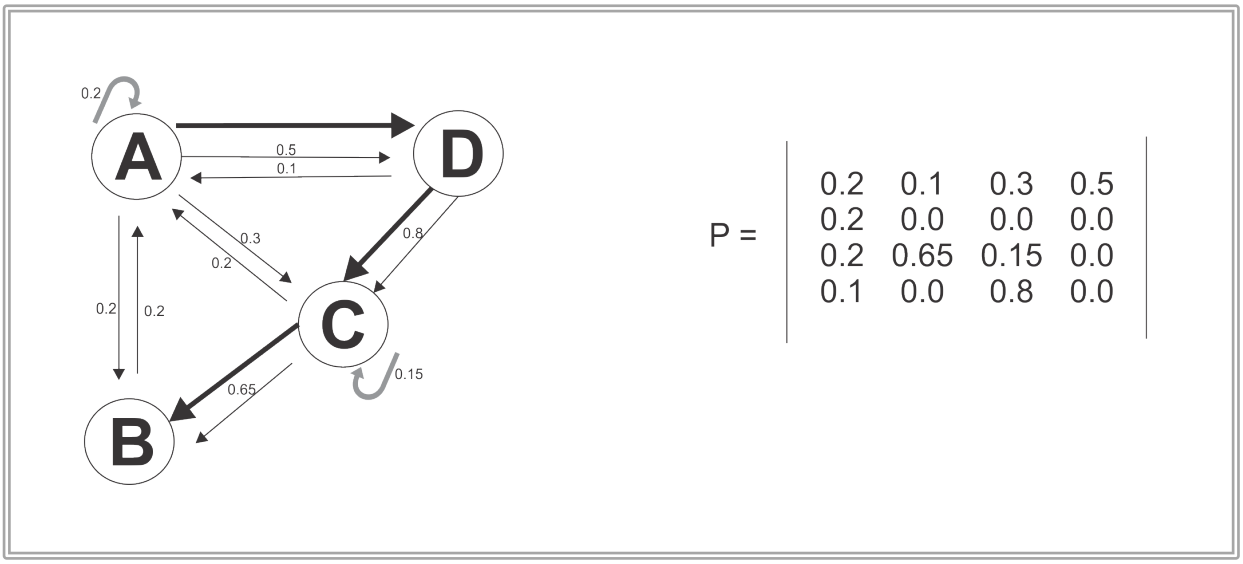


Figura 4 - Funcionamento do modelo com matriz de probabilidades (adaptado de *A Survey of Mobility Models on Wireless Networks*)

em que,

**A** - é a posição inicial do nó
**B, C e D** - são potenciais posições seguintes

Nesta matriz, os valores P (*A*,*B*) representam a probabilidade que o nó tem de ir do estado *A* para o estado *B*. Por exemplo, um nó situado em A tem 0.2 de probabilidade de manter-se na mesma posição, 0.1 de probabilidade de mover-se de A para B e 0.3 de mover-se de A para C. Já para mover-se de A para D, a probabilidade é de 0.5.

Partilhando as características principais do *Random Walk*, este modelo não capta o comportamento realista dos nós em redes móveis sem fio, tomando como exemplo as pessoas que completam tarefas diárias, estas tendem a mover-se num sentido de continuidade do movimento, onde raramente se retoma o movimento contrário ou ocorrem saltos aleatórios, situações que podem acontecer neste modelo de mobilidade.

Nesta medida, a escolha dos valores para P (A,B) torna-se uma tarefa difícil, senão impossível para uma simulação individual, com a exceção de caminhos já pré-definidos para um dado cenário.

#### *2.3.1.3 Random Way-point Mobility Model*

Considerado um derivado do *Random Walk*, este modelo considera um tempo de pausa entre mudanças de direção e velocidade. Inicia com o nó móvel preso numa posição por um determinado período de tempo, *o tempo de pausa*, e após este terminar, é escolhido um destino aleatoriamente na área de simulação e uma velocidade uniformemente distribuída [*minspeed*, *maxspeed*] (Jonhson e Maltz 1996).

Quando são gerados nós lentos e com maiores tempos de pausa, é produzida uma rede muito mais estável do que com nós mais rápidos e tempos de pausa mais curtos. Isto faz com que o *Maxspeed* e o tempo de pausa sejam parâmetros chave do comportamento do nó móvel no modelo.

O facto de que, quando considerada uma área fechada os nós se concentrem no meio da área de simulação, faz com que este modelo não reproduza com realismo as situações de movimentações geográficas.

#### *2.3.1.4 Random Direction Mobility Model (DMM)*

Este modelo surge como resposta aos problemas de concentrações de nós numa dada área da simulação recorrentes do *Random Way-Point*. Deste modo, os nós passam a escolher uma direção aleatória *m*, dirigem-se nessa direção com um movimento semelhante ao do modelo *Random Walk* até chegarem ao limite da área de simulação. Quando atinge o limite da simulação, o nó pára por um tempo específico, escolhe uma direção aposta ao limite, entre 0 e 180 graus e prossegue o processo. A aplicação deste ângulo previne que o nó não ultrapasse o limite de simulação, visto que a direção é limitada.

De importante destaque será a alteração introduzida com o *Modified Random Direction Mobility Model*, possibilitando que os nós continuem a escolher uma direção

aleatória, no entanto, já não são obrigados a deslocar-se até aos limites da simulação antes de poder mudar a direção. Assim, após ter sido escolhida uma direção aleatória, selecionam um destino aleatório e fazem uma pausa no destino antes de escolher uma nova direção aleatória. Esta alteração permite também produzir padrões de movimento capazes de serem simulados com tempos de pausa pelo *Random Walk*.

#### *2.3.1.5 Gauss Markov Mobility Model*

O *Gauss Markov* vem permitir que a velocidade e direções anteriores influenciem a velocidade e direções futuras, possibilitando atenuar as paragens bruscas e mudanças de direção impossíveis verificadas no *Random Walk Mobility Model*. Deste modo, a velocidade e direção no instante *n*, é calculado com base nos valores de velocidade e direção no instante *(n-1)*. Em cada intervalo de tempo, a posição seguinte é calculada com base na velocidade e direção anteriores, sendo então a próxima velocidade e direção escolhidos aleatoriamente na distribuição Gaussiana.

Outra modificação vem assegurar que o nó não se mantenha muito tempo perto do limite da área de simulação, forçando o nó a sair de perto do limite quando este se move a uma certa distância do mesmo. Deste modo, é alterado o ângulo de direção atual para um ângulo oposto ao do limite.

#### *2.3.1.6 Weighted Waypoint Mobility Model*

Este modelo baseado no *Random Waypoint*, veio corresponder ao objectivo de aprofundar a modelação do movimento, nomeadamente, o impacto das escolhas de um nó em direção ao destino. São de salientar, no entanto, as importantes diferenças neste modelo como a posição destino não ser escolhida de forma aleatória. A escolha dos diferentes destinos depende da localização e tempo atual e da diferente distribuição de tempo de pausa nos diferentes sítios (é considerada uma propriedade da localização). A velocidade do nó também não é definida aleatoriamente mas sim definida pela velocidade anterior.

De salientar a capacidade do modelo em capturar o tempo e a posição dependendo dos pesos atribuídos, bem como a possibilidade de configurar diferentes probabilidades de transição, baseado num tempo específico e também consoante os valores do peso atribuído ou popularidade de um local. Dependendo do local em que o nó se encontra e a hora, ele terá comportamentos específicos, por exemplo, se forem meio-dia e um nó que esteja a ser modelado na zona em que foi definida como área de alimentação, irá permanecer por um período de tempo, de modo a simular que o nó parou ali para almoçar.

#### *2.3.1.7 City Section Mobility Model*

O *City Section* proporciona movimentos próximos da realidade de uma secção da cidade uma vez que restringe fortemente o comportamento de deslocação dos nós móveis. Para todos os nós é atribuído um caminho pré-definido e orientações de comportamento, uma vez que na realidade os nós móveis não se podem deslocar livremente sem terem de se preocupar com obstáculos e regras de trânsito.

Posto isto, inicialmente os nós são colocados aleatoriamente na interseção de ruas. A cada rua são associados parâmetros utilizados para determinar o tempo que demora a percorrer a rua, como um limite de velocidade e um determinado comprimento de rua. O nó escolhe aleatoriamente a direção de um cruzamento de uma rua e o algoritmo do movimento da posição corrente para a nova posição procura o percurso mais curto em termos de deslocação do nó. Também fatores correspondentes a uma condução segura, tal como, o limite de velocidade e a distância mínima entre nós, são considerados. Quando o nó atinge o seu destino, pode ser considerado um tempo de pausa e logo após, o processo recomeça novamente.

Como parâmetros a melhorar, pode considerar-se a introdução de tempos de pausa em algumas interseções ou destinos, incorporar variáveis de aceleração e desaceleração e ter em conta um maior ou menor número de nós dependendo da hora do dia. Também a possibilidade de uma maior área de simulação, um aumento do número de estradas, uma via rápida ao longo dos limites da simulação, seriam factores a considerar.

#### *2.3.1.8 Manhattan Mobility Model*

Este modelo é similar ao *City Section*, utiliza uma abordagem probabilística nos movimentos dos nós, ou seja, em cada cruzamento um veículo pode escolher e manter um movimento na mesma direção com uma probabilidade de ½ e virar à esquerda ou à direita com a probabilidade de ¼ para cada um dos casos. Deste modo, os nós são colocados em interseções das ruas, e aqui reside a diferença entre o *City Section*, pois é possibilitada a escolha da direção por parte do nó. A velocidade é sempre precedente, e limitada à velocidade da rua e também dos nós que vão à frente.

#### *2.3.1.9 A Simple Human Mobility Model*

Considerado semelhante ao *City Section* e ao *Manhattan*, este modelo pretende a introdução de estudos comportamentais sociais em modelos de mobilidade individual. Baseado num cenário de ambiente interior, é feita uma divisão entre os nós conduzidos por humanos e os nós estabelecidos previamente em localizações específicas. Deste modo, distinguem-se os nós no interior dos edifícios e os nós inseridos no mapa, partilhando interesses comuns em cada grupo. O movimento de cada nó é modelado segundo uma malha específica, estabelecendo que este modelo seja baseado em dados de movimento já obtidos em estudos anteriores.

### *2.3.2 Modelos de Mobilidade em Grupo*

Nos modelos de mobilidade em grupo todas as movimentações dos nós móveis da simulação são dependentes entre si. Os movimentos padrão destes modelos envolvem um grupo de nós e não apenas um nó individual. Mas neste tipo de modelos, apesar de serem simulados grupos, existe a preocupação de simular os nós individualmente, ou seja, cada nó poderá ter um comportamento diferente dos outros nós, mas andará sempre à volta de um ponto de referência usado pelo grupo.

Nos modelos de mobilidade em grupo mais básicos, existe apenas a preocupação de criar um ponto de referência para o grupo e fazer os nós deslocarem-se de acordo

com esse ponto de referência. Os modelos mais recentes têm começado a especificar cada nó individualmente, um dos conceitos mais usados e que mais se verifica hoje nos tempos atuais, os nós movimentarem-se através das suas afinidades sociais.

#### *2.3.2.1 Reference Point Group Mobility Model*

Funcionando com pontos de referência (centro lógico), este modelo é caracterizado por um movimento de grupo de nós, no qual, cada um possui um espaço individual de movimentos aleatórios e no qual se move de acordo com o seu ponto de referência. Este centro lógico calcula o movimento do grupo através de um vector de movimento (GM), caracterizando a sua direção e velocidade. O movimento dos nós individuais é feito de forma aleatória em torno do seu ponto de referência, dependendo do movimento do grupo.

Assim sendo, inicia-se por o ponto de referência se mover do tempo *t* para o *t+1* e a sua localização atualiza-se de acordo com o centro lógico do grupo. Após a atualização do ponto de referência, *RP (t+1)* é calculado e são combinados com o vetor dos movimentos aleatórios de cada nó sobre o seu ponto de referência individual.

#### *2.3.2.2 Nomadic Mobility Model*

Baseado em estudos comportamentais de comunidades nómadas, este modelo representa movimentos de grupos de nós que, colectivamente, se deslocam de um ponto para outro, estando definido para um ponto de referência um grupo de nós. Estruturalmente, funciona com um ponto de referência e restantes nós, cada um com o seu espaço individual limitado e onde se movem de forma aleatória. Assim, quando o ponto de referência se desloca, os nós do grupo deslocam-se para a nova posição, gravitando em torno do ponto de referência. Este movimento realizado pelos nós é definido através do *Random Waypoint* e a sua velocidade está limitada a um intervalo [*velocidade mínima, velocidade máxima*].

#### *2.3.2.3 Column Mobility Model*

Este modelo também baseado em comportamentos de comunidades, difere do *Nomadic* na medida em que, para cada ponto de referência corresponde um nó individual e o movimento é feito em grupo mas em linha recta (ou em coluna), permitindo que os nós individuais possam seguir-se uns aos outros. Deste modo, é definida uma grelha inicial e cada nó individual tem apenas um ponto de referência, movendo-se aleatoriamente em torno deste através do Random Walk. O movimento deste modelo é calculado através de apenas uma equação (*Novo ponto de referência = antigo ponto de referência + vector avançado*), na qual o ponto de referência é escolhido pela soma do antigo ponto de referência do nó e o vector avançado (offset pré-definido que se move na grelha de referência).

#### *2.3.2.4 Pursue Mobility Model*

Neste modelo é possível a simulação de uma perseguição a um ponto de referência por outros nós em grupo, de tal forma que, a direção e velocidade são baseados nos movimentos do ponto a perseguir. Assim, o modelo desenvolve-se a partir de apenas uma equação que permite o cálculo da nova posição de cada nó, resumindo-se na soma da antiga posição, da aceleração e do vetor aleatório relativo a cada nó (obtido através de um *Random Walk*): *Nova posição = antiga posição + aceleração (alvo antiga posição) + vector aleatório*.

Também de salientar, será o facto de que o limite de aleatoriedade de cada nó é reduzido de forma a aumentar a eficiência na perseguição ao ponto de referência.

#### *2.3.2.5 Community Based Mobility Model*

Este modelo também influenciado por comportamentos sociais, é baseado numa função de atratividade social, no qual os nós estão agrupados em comunidade e movem-se em direção uns aos outros e até em redor de outras comunidades. Esta atratividade é calculada com base em probabilidades e força de associativismo entre nós, sendo a

representação de interação entre nós, de igual força social entre o nó *i* e *j*, e entre o nó *j* e *i*. A configuração inicial dos valores de força social entre nós é feita através de algoritmos, fornecendo uma medida de centralidade entre os mesmos. Os nós caraterizados por valores altos de centralidade são detetados e removidos, de forma a extrair as comunidades da rede.

O movimento dos nós é definido pelas associações que um nó tem num determinado instante de tempo e a atratividade social é definida por duas abordagens, a determinista e a probabilística. Na primeira, o novo alvo é escolhido aleatoriamente dentro de uma área considerada de maior atratividade social e a partir daí, o nó move-se em direção a uma comunidade existente e nunca a uma célula vazia. Na segunda, o alvo é escolhido proporcionalmente à atratividade de cada comunidade.

#### *2.3.2.6 Home Cell Community Mobility Model*

Este modelo também é baseado em aspectos sociais, sendo que a principal diferença é que este modelo não considera apenas a atração a outros nós, mas também considera a atração a localizações específicas. Deste modo, são aplicadas estatísticas com preferências por distâncias mais curtas em vez de longas, e o modelo assume que para cada nó existe uma *home cell,* na qual o nó é colocado inicialmente e tem uma atração maior do que as outras. Colocado o nó na sua *home cell* ou célula de origem, o próximo alvo é calculado como na abordagem probabilística do *Community Mobility Model*, através da atração dos nós e acrescentando ainda a atração das localizações. Do mesmo modo, quando o nó está numa célula que não a de origem, a probabilidade de se manter nessa célula é definida por *p* e a probabilidade de retomar a sua célula de origem é de *1-p*.

## 2.4 Projetos Relacionados

Pretende-se com esta secção, fazer uma análise de simuladores semelhantes ao que será desenvolvido, não apenas de simuladores open-source como o *BonnMotion*, o

*The One* e o *SUMO*, mas também foi necessária a avaliação de um simulador não open-source, como o *VISSIM*.

### *2.4.1 BonnMotion*

O *BonnMotion* é uma aplicação *Java* utilizada para criar e analisar os cenários de mobilidade. Desenvolvido pelo grupo de Sistemas de Comunicação da Universidade de Bonn, na Alemanha, serve de ferramenta na investigação de dispositivos móveis. Como software *open-source*, o *BonnMotion* encontra-se atualmente na versão 2.0 lançada a 07/11/2011 e pode ser feito o download no site do projeto (http://net.cs.uni-bonn.de/start/ , acedido a Outubro de 2012).

O simulador *BonnMotion* permite gerar diferentes tipos de cenários de mobilidade que têm como base modelos de mobilidade já descritos. De salientar, alguns modelos suportados no simulador:

- Random Waypoint Mobility
- Manhattan Grid Mobility
- Gauss-Markov Mobility (original)
- Gauss-Markov Mobility
- Reference Point Group Mobility
- Disaster Area
- Self-similar Least Action Walk
- Random Direction Mobility
- Random Walk Mobility
- Column Mobility
- Nomadic Community Mobility

O *BonnMotion* não tem ambiente gráfico, sendo este programa executado na linha de comando. Existem duas maneiras para definir os parâmetros de entrada na geração de um cenário. A primeira é introduzir os parâmetros na linha de comando e a segunda é ter um ficheiro com os parâmetros. Os dois métodos podem ser combinados e os

parâmetros introduzidos na linha de comando podem substituir os parâmetros introduzidos no ficheiro.

Entre os parâmetros configuráveis destacam-se o número de nós, duração da simulação, tempo inicial da simulação que se pretende ignorar, tamanho da área de simulação, qual o modelo de mobilidade que se pretende simular, velocidade máxima e mínima e tempos de pausa.

Após a simulação são criados dois ficheiros, um com a *extenção .params* e o outro com a *extenção .movements*. O primeiro ficheiro vai conter todos os parâmetros definidos no início da simulação. O segundo ficheiro vai ter as coordenadas correspondentes aos movimentos de cada nó criado.

Na *Figura 5* pode-se ver um exemplo do ficheiro *.movements*, onde se simulou dois nós móveis. No primeiro valor do documento é o tempo inicial da simulação, o segundo e terceiro valores são a posição inicial do nó em x e y respetivamente, o quarto é o tempo final da simulação, o quinto e sexto são os valores da posição final do nó. Os tempos a seguir começam na posição em que ficou e repete-se a mesma estrutura dos primeiros campos até ao fim da simulação.

```
0.0 57.00041320614643 105.4061130367185 101.10329898858254 123.57093897035567 105.4061130367185 145.2999070302380
123.57093897035567 105.4061130367185 154.3904198567152 123.57093897035567 115.49655190951454 163.790735497666
123.57093897035567 115.49655190951454 202.52101659823575 123.57093897035567 88.52308875320335 209.56974633113123
123.57093897035567 88.52308875320335 375.81446250336467 10.312224212476018 88.52308875320335 379.824327833639
10.312224212476018 88.52308875320335 470.1857074565032 10.312224212476018 199.80784346872105 507.34416875281477
10.312224212476018 199.80784346872105 608.7365144382838 154.35159394621803 199.80784346872105 609.64836549285543
154.35159394621803 199.80784346872105 800.6138159848924 154.35159394621803 6.275621734980907 815.7764420381809
154.35159394621803 6.275621734980907 902.9845907090721 154.35159394621803 51.41013488189863 958.8064139018325
154.35159394621803 51.41013488189863 1000.0 154.35159394621803 94.8150010545168

0.0 109.90886907460303 93.20141825410802 66.63056926116133 50.678332323575106 93.20141825410802 99.63538527156697
50.678332323575106 93.20141825410802 189.3824546013807 168.85328250673402 93.20141825410802 226.25402757873962
168.85328250673402 93.20141825410802 279.1490439893114 168.85328250673402 59.298460131876226 315.51768894547513
168.85328250673402 59.298460131876226 451.80640621858356 59.48516959510266 59.298460131876226 476.7488119887198
59.48516959510266 59.298460131876226 478.01380495685225 59.48516959510266 60.837549656048395 521.0858881079375
59.48516959510266 60.837549656048395 540.9673871116493 87.82938668937736 60.837549656048395 553.0405237873874
87.82938668937736 60.837549656048395 578.6458553795064 116.28245889916424 60.837549656048395 625.9519125689358
116.28245889916424 60.837549656048395 676.8963906883437 116.28245889916424 7.532681185922119 735.7161620463467
116.28245889916424 7.532681185922119 812.3976894734942 116.28245889916424 111.8881487496362 856.3213337112547
116.28245889916424 111.8881487496362 970.6565751738854 116.28245889916424 177.81090578858849 975.1785590161662
116.28245889916424 177.81090578858849 994.5662008582203 92.51222461878568 177.81090578858849 996.8259070861159
92.51222461878568 177.81090578858849 1000.0 92.51222461878568 176.18131671100963
```

Figura 5 - Estrutura do ficheiro movements

Os cenários criados por este simulador podem ser exportados para outros simuladores de redes para que sejam usados por esses mesmos simuladores, nomeadamente pelos seguintes:

- NS-2
- GloMoSim/QualNet
- COOJA
- MiXiM
- The ONE

Além deste formatos, o *BonnMotion* permite ainda que se guarde a informação em XML.

Como esta aplicação não possui qualquer tipo de ambiente gráfico para acompanhar ou analisar a simulação, existe a possibilidade de se exportar um ficheiro que permite seguir a movimentação de um dos nós usando o *gnuplot* (*http://www.gnuplot.info/, acedido a Outubro 2012).*

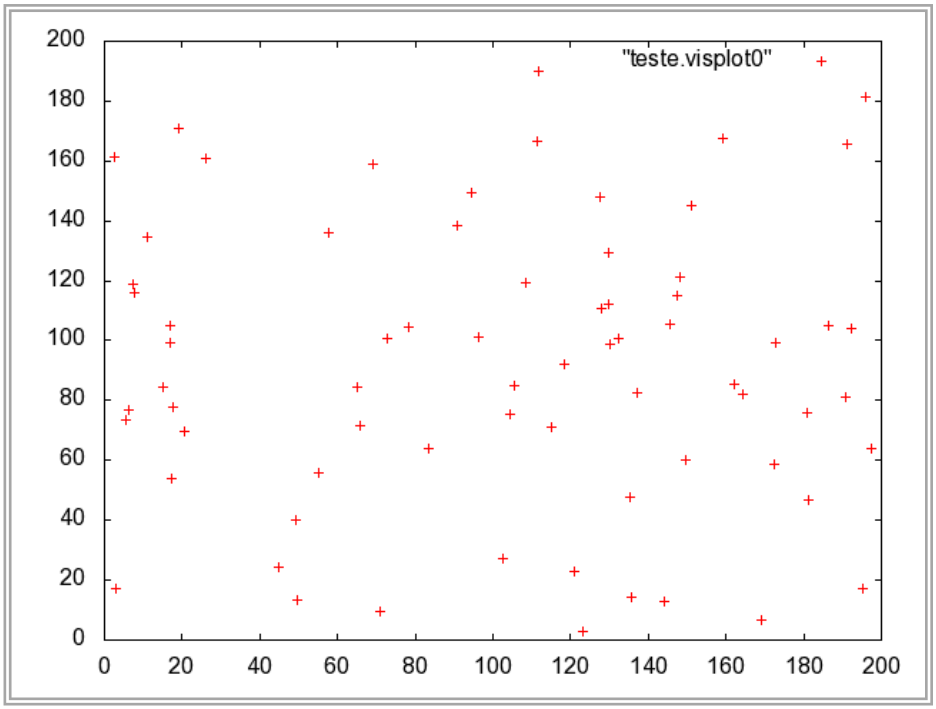


Figura 6 - Exemplo de um cenário de teste visto no gnuplot

Na *Figura 6* podemos ver o resultado final de uma simulação teste efectuada no BonnMotion e de seguida visualizada com o *gnuplot*. Os eixos do gráfico representam o *x* e o *y*, a área de simulação foi definida em x=200 e y=200. No cenário estão representados os pontos em que um nó esteve ao longo do tempo de simulação.

### *2.4.2 The One*

O *The ONE (Opportunistic Network Environment)* é um simulador de redes oportunísticas. O simulador está a ser desenvolvido pelos projetos SINDTN (*Security Infrastructure for Delay Tolerant Network*) (SINDTN 2011) e CATDTN (*Connectivity, Applications, and Trials of Delay Tolerant Networking*) e conta com o apoio da *Nokia Research Center* (*Finland*) (Nokia Research Center 2011). O simulador está a ser desenvolvido em Java e é um projeto *open-source*, estando o seu código disponível para *download* no site do projeto (http://www.netlab.tkk.fi/tutkimus/dtn/theone/, *acedido a Outubro 2012*). Ao nível da portabilidade, sendo o simulador uma aplicação em *Java*, o The ONE é bastante flexível uma vez que funciona sem problemas nos principais sistemas operativos da atualidade, *Windows, Mac OSX* e *Linux*. A última versão lançada do software foi a 1.4.1 e foi lançado em 31/01/2011.

O simulador suporta nós móveis com diferentes modelos de mobilidade, troca de mensagens entre os nós, vários algoritmos de *routing* e possui um interface gráfico que permite visualizar os nós em movimento e a troca de mensagens entre os mesmos em tempo real (Keränen, Ott e Kärkkäinen 2009).

O ambiente gráfico do simulador é simples e intuitivo. A janela do simulador representada na *Figura 7,* é composta por uma barra lateral onde é possível ver e ter acesso aos nós que estão ativos (Nodes), no fundo tem as ligações que os nós vão efetuando (Event log) e no centro está a área de simulação onde estão representados o mapa e os nós. Na barra de cima encontram-se os controles da simulação onde é possível fazer captura de imagens, parar a simulação, acelerar ou abrandar a simulação e ainda fazer zoom da área de simulação.

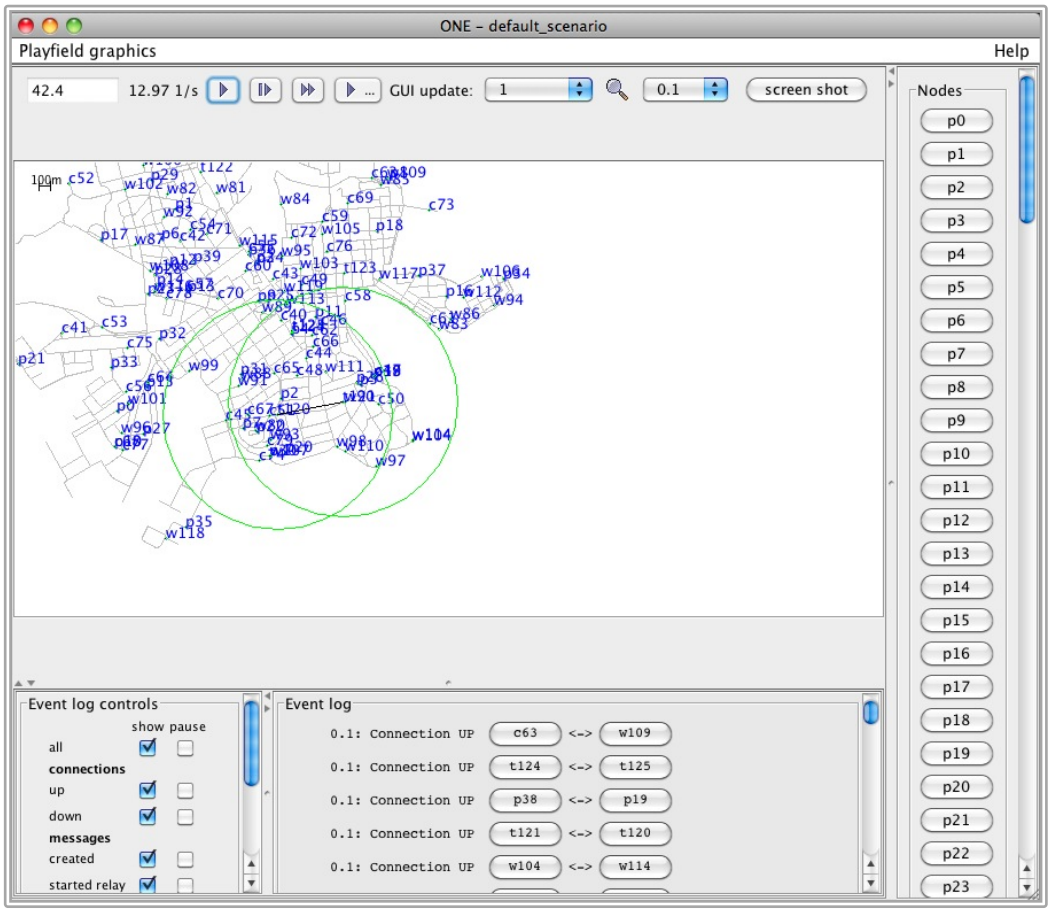


Figura 7 - Ambiente gráfico do The One (Silva 2011).

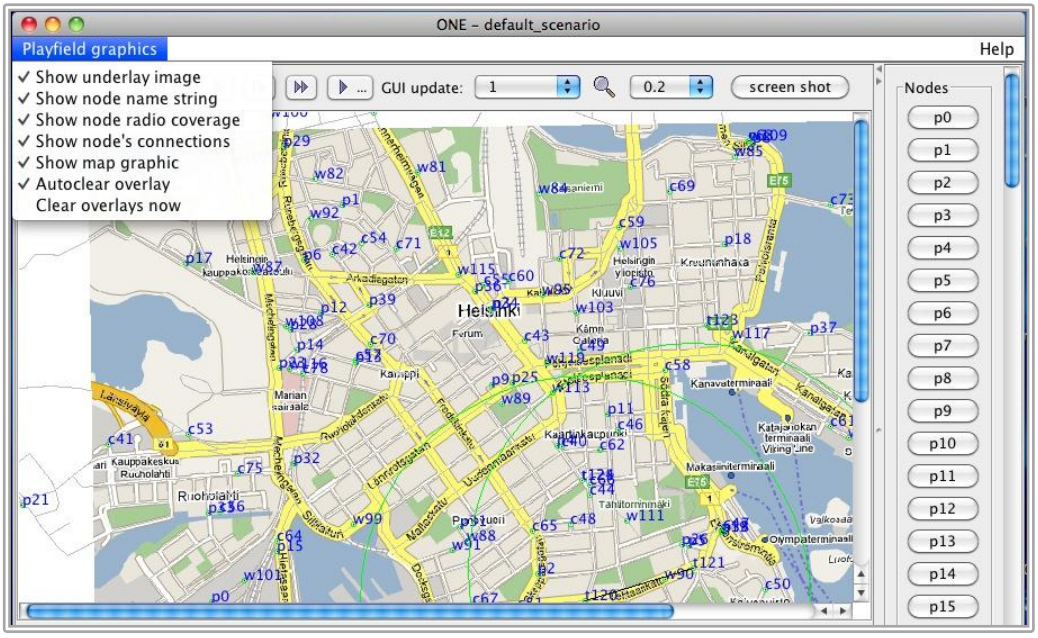


Figura 8 - Ambiente gráfico com o mapa como fundo (Silva 2011).

Como se pode ver na *Figura 8*, no canto superior esquerdo do ecrã aparece um menu com o nome de *Playfield graphics*. Se se abrir esse menu aparecem várias opções de visualização na janela que contém os nós e os seus caminhos, como é o caso do mapa base que foi usado para construir o percurso apresentado, neste caso um mapa do centro de Helsínquia.

Para a construção de um novo mapa, a ser usado pelo The One, é necessário usar uma outra ferramenta, o OpenJUMP. Este software é também open source e possibilita a criação de mapas vectoriais no formatowkt(Well-Known Text). É uma ferramenta que pode ser usada nos principais sistemas operativos (*Windows, Mac OSX, Linux*). É uma forma muito simples para gerar mapas bastante realistas.

O OpenJUMP permite, no mesmo projeto, criar diversas layers, sendo assim possivel diferentes nós terem comportamentos diferente nas mesmas zonas do mapa. Por exemplo, num mapa de uma cidade temos ruas destinadas para os pedestres, estradas destinadas para veículos, estradas só para transportes públicos ou até ruas para todos eles. Se for criada uma layer para cada interveniente, em cada layer vai resultar um mapa incompleto, mas se todas as layers forem colocadas umas em cima das outras vão dar origem ao mapa completo. Cada layer, no fundo, é uma pequena parte do mapa completo, em que restringe os nós a deslocarem-se apenas nas ruas ou estradas destinadas para eles.

As grandes vantagens do simulador *The ONE* é o facto de ser multiplataforma, permitir que sejam carregados diferentes mapas, simular o movimento de diversos tipos de nós assim como as suas interações. Tem um ambiente gráfico bastante intuitivo (*Figura 9*) e vai mostrando ao utilizador, em tempo real, quando é que os nós se conectam e desconectam. Além disso, permite que o utilizador controle que tipo de *logs* pretende ver durante a simulação, sendo que os pode alterar a qualquer momento. Uma outra característica muito relevante no *The ONE* é o facto de se poder acelerar ou abrandar a velocidade da simulação. Por fim, podemos ainda referir que este simulador é multiplataforma e permite que sejam carregados diferentes mapas.

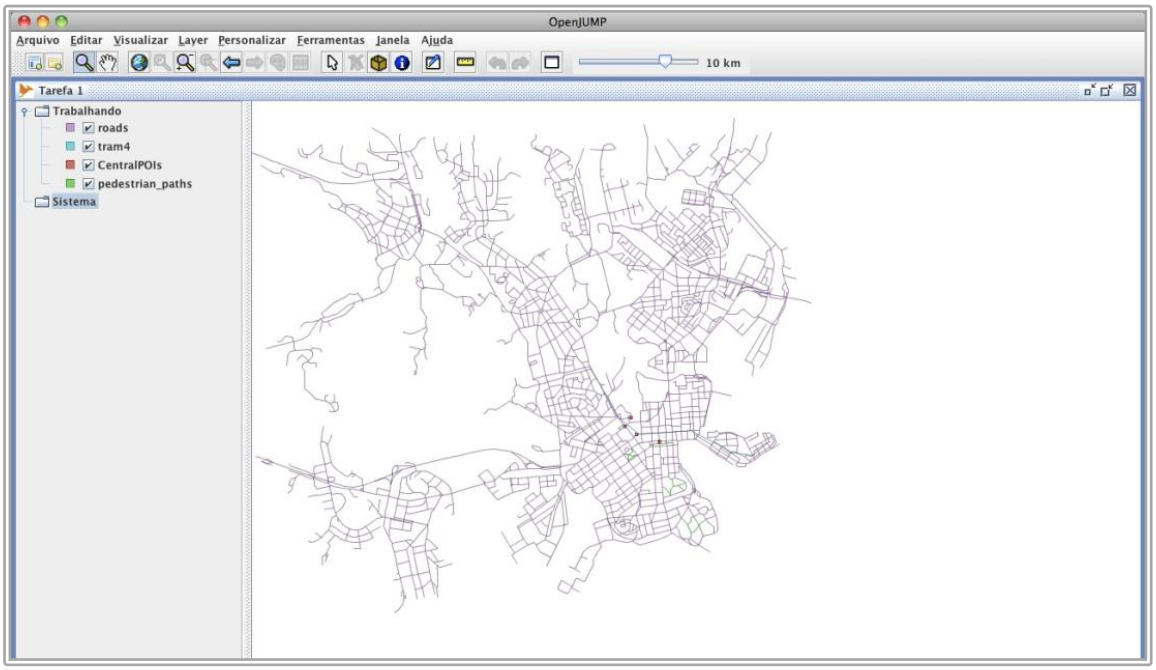


Figura 9 - Ambiente gráfico do openJUMP (Silva 2011).

Mas o The ONE possui limitações como, por exemplo, o facto de a quantidade de atores durante a simulação ser sempre o mesmo, e os atores não possuírem comportamentos de mobilidade variados, isto é, os seus movimentos fazem sempre o mesmo circuito e com a mesma velocidade.

### *2.4.3 SUMO*

O desenvolvimento do *SUMO (Simulation of Urban MObility)* foi iniciado no ano de 2000. É um software open source, desenvolvido maioritariamente pelos funcionários do Institute of Transportation Systems no centro aeroespacial alemão. O SUMO tem como principal objetivo dar apoio à comunidade de pesquisa de tráfego com uma ferramenta que permite implementar e avaliar os seus próprios algoritmos sem ser preciso ter a preocupação em criar ou implementar métodos para lidar com as estradas e controles de tráfego.

O *SUMO* é desenvolvido na linguagem C++ e pode ser executado nos sistemas operativos Linux e Windows. Ele permite simular trânsito urbano, com a possibilidade

de criar mapas, veículos de vários tipos, cada um movendo-se individualmente e com um percurso próprio, permite também controlo e gestão do fluxo de trânsito.

Algumas características de simulação do SUMO são:

- Movimento de veículos em espaços contínuos e tempos discretos
- Diferentes tipos de veículos
- Várias linhas de estradas com mudança de linha
- Diferentes prioridades na estrada e semáforos
- Consegue gerir cenários com grandes quantidades de ruas.
- Boa performance com um grande número de veículos
- Interoperabilidade com aplicações em tempo real

Podem ser importados diversos formatos de outros softwares tais como *PTV VISUM*, *PTV Vissim*, *shapefiles*, *Open Street Map* (OSM), *RoboCup*, *MATsim*, *openDRIVE* e ficheiros XML. A última versão do SUMO a 0.14.0 foi lançada em 11/01/2012.

Para se criar uma simulação é necessário criar quatro ficheiros XML, um para definir pontos que constituem as estradas (extensão *.nod.xml*), um para definir os sentidos de circulação (extensão *.edg.xml*), um outro que define os percursos pretendidos assim como as viaturas (extensão *.rou.xml*) e, por fim, um ficheiro gerado pelo comando "NETCONVERT" (extensão *.net.xml*). Este último importa os ficheiros onde estão as estradas e os sentidos de circulação e retorna um outro ficheiro onde se pode controlar os limites de velocidade e o tipo de veículos que irão circular durante a simulação.

Para efetivamente ocorrer um exemplo de simulação, é necessário criar um outro ficheiro com a extensão *.cfg*, onde indicamos os ficheiros *.net* e *.rou* para o *SUMO* saber o que executar. É também neste ficheiro que se define a duração da simulação.

Após todos os ficheiros terem sido criados, é possível executar o programa e carregar os ficheiros, iniciando assim a simulação. Na *Figura 10* é possível visualizar um exemplo de um cenário de simulação.

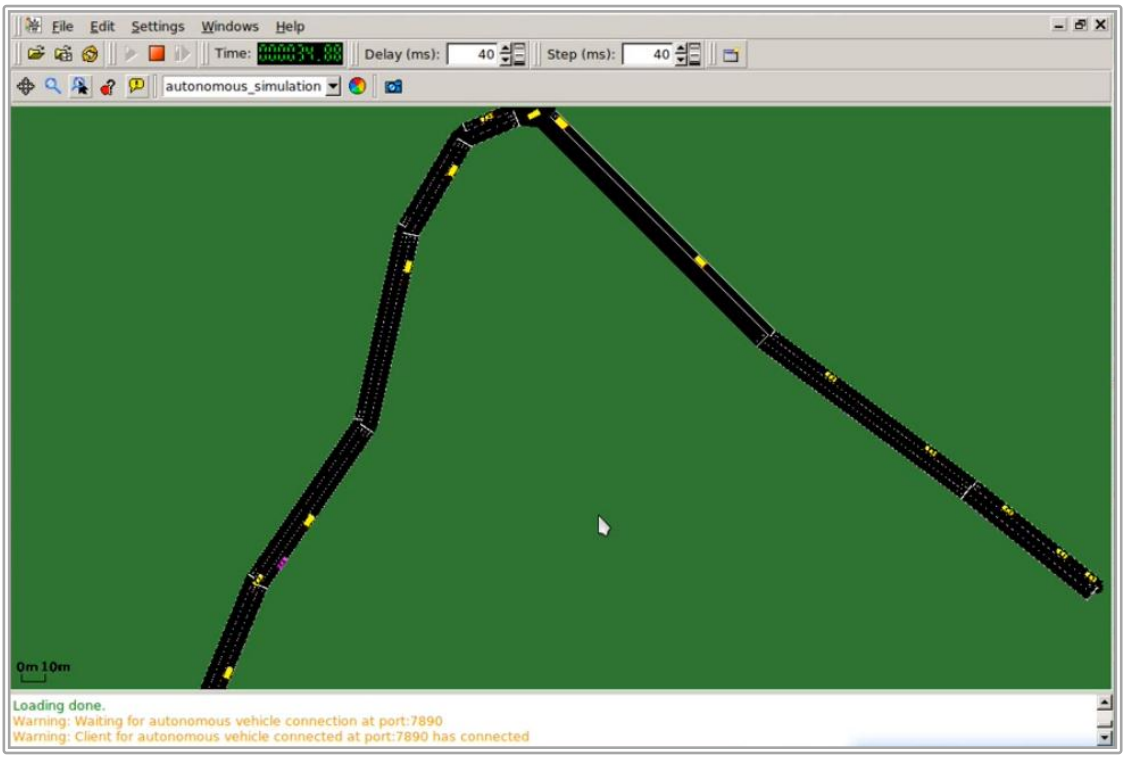


Figura 10 – Ambiente gráfico do SUMO.

### *2.4.4 Vissim*

O VISSIM (*Verkehr In Stadten-SIMmulation*, que significa simulação de tráfego em áreas urbanas) é um modelo de simulação de tráfego microscópico estocástico capaz de representar o comportamento do tráfego rodoviário. Começou a ser desenvolvido na Alemanha, na Universidade de Karlsruhe, na década de 70. A sua distribuição comercial começou em 1994 sendo a sua versão mais recente a 5.40 lançada em 2011. Atualmente o Vissim é líder no mercado dos simuladores de tráfego.

O simulador é baseado num modelo microscópico de tráfego discreto e estocástico, ou seja, é baseado em diversos modelos matemáticos e as posições de cada entidade é recalculada em intervalos de tempo entre 0,1 segundos e 1 segundo. O software foi implementado em *C++*, com uma programação orientada a objectos. Em cada classe os objectos (por exemplo veículos) são caracterizados por atributos e métodos que descrevem as funções. O simulador apenas está disponível para o sistema operativo *Windows*.

O VISSSIM pode ser usado para modelar transportes públicos e privados, assim como movimentos pedonais (Barceló 2010). É usado para fazer o estudo, planeamento e avaliação de diversos cenários. O simulador é capaz de simular áreas de diversas dimensões e mesmo algo mais específicas como apenas uma rua, um cruzamento, uma rotunda ou uma paragem de metro.

O simulador é normalmente usado para:

-Realizar estudos prévios relacionados com o ordenamento do território.
-Criar estratégias para implementação de novas estradas principais e secundárias.
-Estudo do impacto ambiental.
-Criar sistemas de controlo de fluxo de tráfego em zona de grande congestionamento.
-Realizar um estudo da implementação de um novo transporte numa dada cidade.
-Avaliação de sistemas de transportes inteligentes.

É possível simular diversas entidades, tais como carros, autocarros, motociclos, bicicletas, pedestres, comboios e semáforos. Em 2006, implementaram no simulador a funcionalidade de usar multiprocessadores de um computador e de usar processamento distribuído num cluster de computadores, com o intuito de diminuir o tempo computacional de uma simulação de uma grande rede. O VISSIM consegue criar simulações com um número elevado de entidades numa só simulação (segundo a PTV, 30.000), e com uma simulação distribuída pode chegar as centenas de milhares (segundo a PTV,300.000).

No simulador VISSIM, sendo baseado num modelo microscópico, o tráfego é modelado com muito detalhe. Para conseguir simular com mais detalhe, o simulador foi divido em três blocos de construção mais importantes e um bloco adicional que gere os dados de saída da simulação (*Figura 11*).

O bloco *Infraestruturas* inclui a modelação de tudo que é considerado físico na simulação, como estradas e as vias ferroviárias, as paragens e parques de estacionamento dos transportes públicos, sinalização, detectores de velocidade, etc..

As características técnicas dos veículos e as especificações do fluxo de tráfego são feitas no bloco *Tráfego*. Todos os elementos de *Controlo do Tráfego*, como semáforos,

regras a que as entidades precisam de obedecer, características e definições das sinalizações, etc., pertencem ao bloco Controlo.

Todos os blocos têm uma dependência entre eles, representada pelas setas bidirecionais. Durante a simulação, os blocos são constantemente ativados pelas interdependências entre eles.

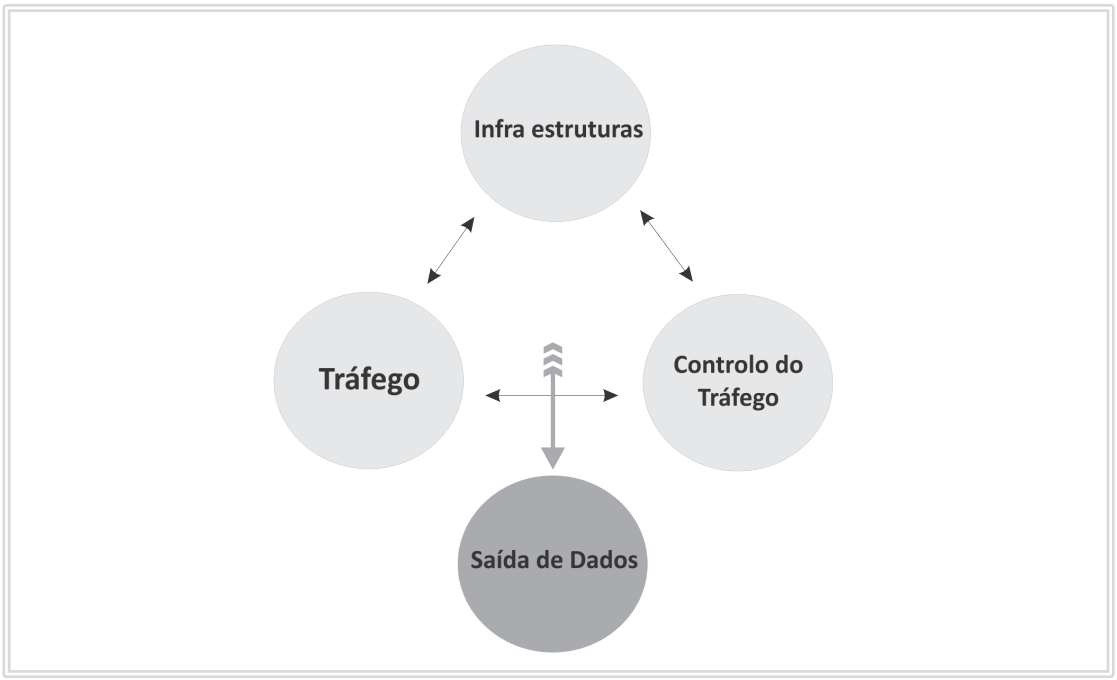


Figura 11 - Arquitetura geral do VISSIM (adaptado do VISSIM 5.10 User Manual).

No caso do bloco *Saída de Dados,* apenas recebe os dados dos outros três blocos, sem qualquer influência no seu funcionamento. Os dados gerados durante a simulação podem chegar ao utilizador como uma animação dos veículos e do estado dos controladores de tráfego, ou podem ser apenas dados estatísticos dos estados dos veículos, num ficheiro. Os valores mais comuns que podem resultar da simulação são o atraso, velocidade, densidade, tempo de viagem e comprimentos das filas de espera.

O VISSIM possui alguns modelos fundamentais implementados no núcleo das suas entidades, e estes representam o comportamento que as diversas entidades vão ter. Os modelos mais relevantes usados pelo VISSIM são:

- *Car following*
- *Lateral movement*
- *Tactical driving behaviour*
- *Pedestrian modeling*
- *Fixed route*
- *Dynamic route*

Graficamente, o VISSIM é capaz de apresentar o ambiente de simulação em 2 e 3 dimensões. Permite ainda que os utilizadores criem pequenos vídeos a demonstrar as simulações, guardando o ficheiro em formato *AVI*. Possui um interface para o utilizador criar o seu próprio cenário (*Figura 12*), e é possível importar fotografias e ficheiros do *AutoCAD* para usar como imagem de fundo nos cenários. Consegue também importar desenhos 3D do programa Google Sketchup para incluir na simulação e exportar a simulação para o programa *AutoDesks 3DSMAX* para criar uma visualização mais realista da simulação (*Figura 13*).

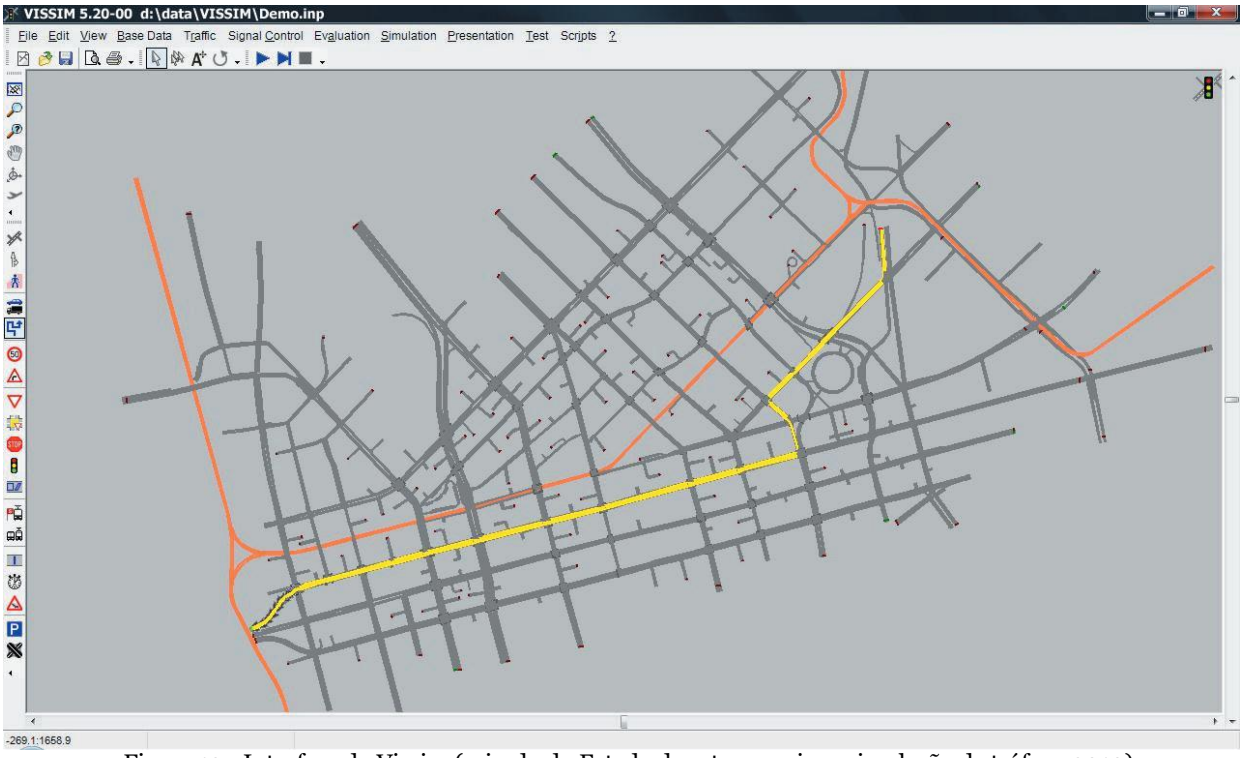

Figura 12 - Interface do Vissim (reirado do Estado da arte em micro-simulação de tráfego 2010)

Figura 13 - Visualização de um cenário 3D no Vissim (Barceló 2010)

O VISSIM é compatível com outros programas usados na área de planeamento e implementação de redes de transporte, assim como o *Synchro*, *TEAPAC*, *Scoot*, *Tranplan*, *NAVTEQ*, entre outos (VISSIM - Estado-da-arte em micro-simulação de tráfego, 2010).

O simulador possui ainda um interface de programação (interface COM), que permite aos utilizadores a integração dos seus programas com o VISSIM, aumentando assim a flexibilidade do simulador. As linguagens que o COM suporta são o *Visual Basic*, *Python* e *C++*. Os utilizadores conseguem ter acesso aos controladores de semáforos, rotas de tráfego e comportamento dos veículos, entre outras funcionalidades.

## 2.5 Conclusões

Em resumo, este capítulo tratou de tipos de simulação, modelos de mobilidade e outros simuladores existentes. Uma primeira referência às soluções que adotamos, o

nosso simulador será do tipo agent-driven, as entidades terão como base os modelos de mobilidade como *Random Walk*, o *City Section* e *Manhattan* e, devido ao nível de detalhe que pretendemos dar as entidades, será um simulador microscópico.

Foram identificadas claras limitações nos simuladores existentes que o nosso simulador irá tentar ultrapassar. No caso do simulador Vissim, é um software que é pago, o que faz com que seja um grande entrave para qualquer pessoa ou instituição, o simulador também não permite realizar comunicação de rádio entre os diversos nós. No caso dos simuladores open-source, o The One e o SUMO têm a limitação de criar simulações em grande escala. Já o BonnMotion não possui interface gráfico, não é possível acompanhar a simulação e apenas disponibiliza os resultados da simulação num ficheiro.

Devido às falhas que foram enumeradas nos simuladores estudados, acreditamos que seja possível criar um simulador que seja completo e que consiga colmatar as falhas dos simuladores que foram referidos anteriormente.

# 3 Bartolomeu Urban Mobility Simulator

Este capítulo descreve a arquitetura do simulador que se pretende implementar, de modo a ser possível dar resposta aos requisitos inicialmente definidos, uma vez que os simuladores atualmente existentes não cumprem todos esses requisitos.

O nome a atribuir ao simulador Bartolomeu Urban Mobility Simulator (BartUM) foi inspirado no nome do navegador português Bartolomeu Dias, que foi o primeiro europeu a navegar além do Cabo da Boa Esperança, no extremo sul de África em 1488.

Neste capítulo será apresentada a estrutura do simulador, como foi projetada e implementada a sua solução.

A equipa deste projeto era composta por três elementos e, por isso, a arquitetura do simulador foi dividida em três partes, de modo a atribuir tarefas distintas a cada um dos elementos. As três divisões foram o núcleo, ou *core,* do simulador (Silva 2011), os atores, e a visualização. Os atores serão a parte mais detalhada, pois foi a parte da simulação que me foi atribuída. Todos os elementos do simulador foram desenvolvidos na linguagem *Java*.

## 3.1 Arquitetura do sistema

Na arquitetura geral do simulador destacam-se três blocos, o *GlobalCoordinator*, o *LocalCoordinator* e a *Visualization*. Cada um destes blocos é executado separadamente. Podem ser executados todos na mesma máquina, se a simulação for pequena, ou em máquinas separadas para uma simulação em larga escala. Nesse caso, as máquinas estarão ligadas através de uma rede de área local (LAN) para permitir a troca de informação entre eles. Na *Figura 14* apresenta-se a arquitetura do sistema:

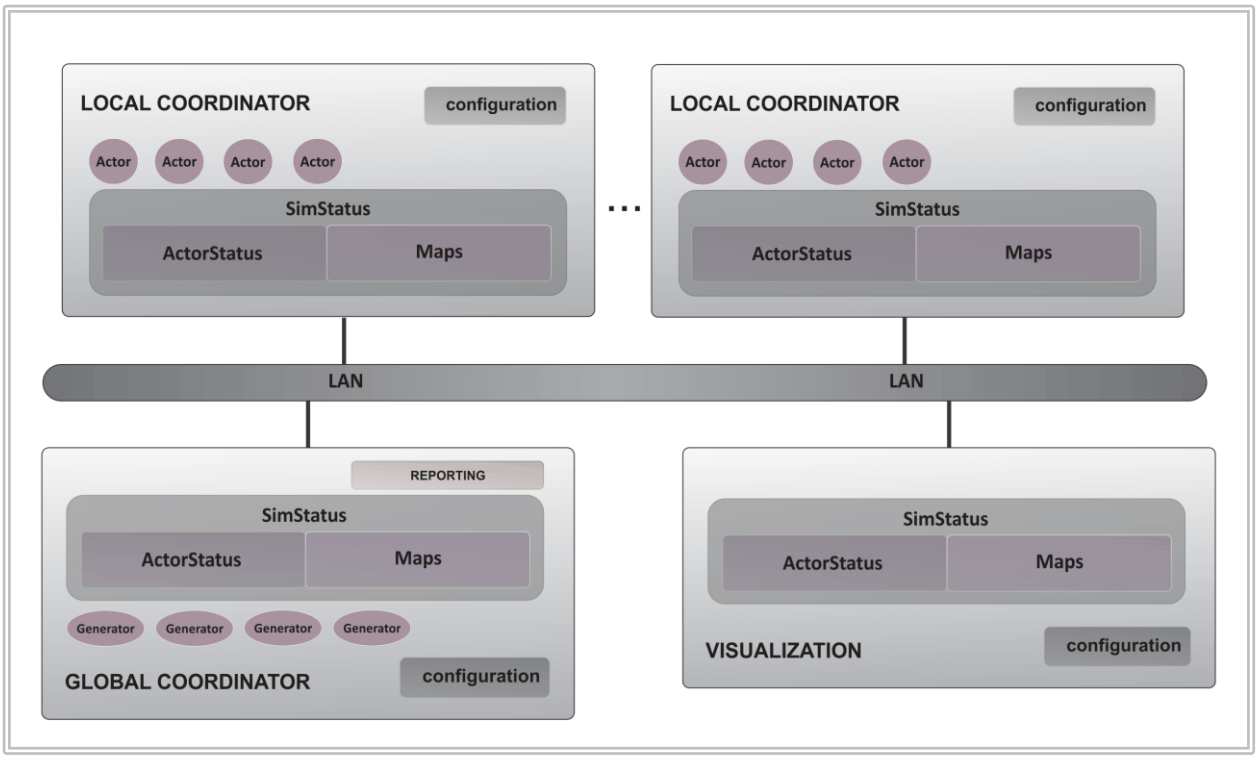


Figura 14 - Arquitetura do sistema

De seguida, será feita uma descrição das entidades mais relevantes do simulador, nomeadamente do *GlobalCoordinator e LocalCoordinator*, dos *Actors* e dos seus respectivos *Generators*, dos *Maps*, da *Visualization* e do *Reporting*.

### *3.1.1 Funcionamento geral do sistema*

O *GlobalCoordinator* é o bloco central do sistema. O sistema é inicializado e controlado a partir desta entidade. É no *GlobalCoordinator* que são tomadas as decisões de criação dos atores e quais os *LocalCoordinators* a que cada um dos atores será atribuído. Como um dos objectivos do simulador é criar simulações em larga escala, o *GlobalCoordinator* vai gerir o modo como os atores serão distribuídos pelos *LocalCoordinators*, de maneira a que a carga se mantenha equilibrada por todos eles. Sendo o elemento de arranque e gestão do sistema, apenas haverá um *GlobalCoordinator* por cada simulação.

O *GlobalCoordinator* integra o *SimStatus*, o *Reporting* e os *Generators*. O *SimStatus* é a entidade que armazena toda a informação relativa ao estado atual da simulação. Será nesta entidade que residirá a informação sobre os mapas que serão usados na simulação e que, em qualquer momento, podem ser requisitados pelo *LocalCoordinator* e pela *Visualization,* e terá a informação sobre a carga de cada *LocalCoordinator* existente. O *Reporting* e os *Generators* serão descritos mais à frente.

O *LocalCoordinator* será muito semelhante ao *GlobalCoordinator* mas funciona como um coordenador local e caberá a ele monitorizar o estado dos atores que estão a ser executados na sua máquina. Poderão existir vários *LocalCoordinators*, dependendo do tamanho da simulação e suas necessidades. Consoante o número de atores necessários, ajusta-se a quantidade de *LocalCoordinators*.

O *LocalCoordinator* é composto pelo *SimStatus* e pelos *Actors*. O *SimStatus* é muito semelhante ao *SimStatus* existente no *GlobalCoordinator*. Os *Actors* serão descritos mais à frente.

Para permitir a ligação dos diferentes blocos em diferentes máquinas foram adotados dois processos diferentes para o envio de dados através da rede que interliga as várias máquinas. Para o envio de mensagens consideradas essenciais e que não podem ser perdidas, como as mensagens do *GlobalCoordinator* para os *LocalCoordinators* com a ordem de criação de atores, ou o envio dos mapas do *GlobalCoordinator* para o *LocalCoordinator* e para a *Visualization*, é usado o protocolo TCP. Com o protocolo TCP é garantida a fiabilidade necessária para o envio de informação mais sensível.

Quando ocorre uma atualização dos Actors de um *LocalCoordinator*, essa informação tem de ser difundida para todas as entidades do sistema, para o *GlobalCoordinator*, para os outros *LocalCoordinators* e para a *Visualization*. Sendo uma informação que é partilhada por todo o sistema, é usada uma abordagem de envio de mensagem para um grupo multicast. Assim, quando existem atualizações, todas as entidades a vão receber, tornando-se vantajoso pelo facto de não haver sobrecarga do *GlobalCoordinator* em receber todas as atualizações e depois as enviar para cada uma das entidades. Estas mensagens são enviadas com base no protocolo UDP. Na *Figura 15*

pode-se ver como foram distribuídos os diferentes modulos de comunição das entidades do sistema e com a devida explicação.

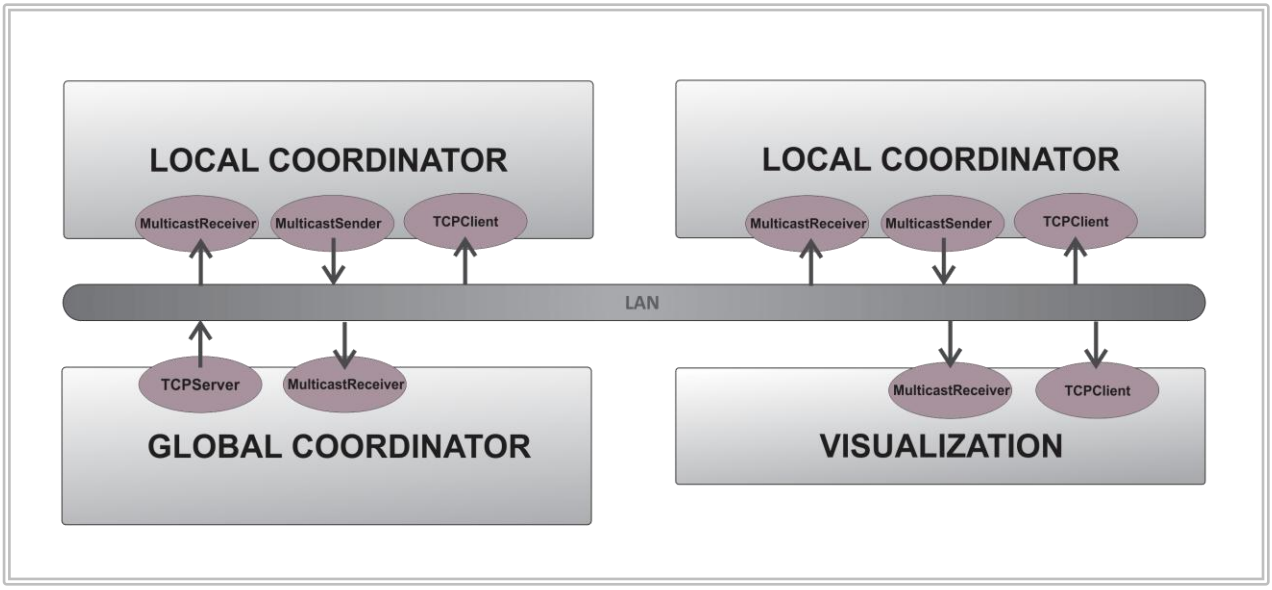


Figura 15 - dispositivos de comunicação na arquitetura

**Multicast Sender:** Usado para o envio de atualizações dos Actors.
**Multicast Receiver:** Usado para a receção de atualizações dos Actors.
**TCP Server:** Usado para enviar as ordens de criação dos Actors e envio de mapas.
**TCP Receiver:** Usado para receção das ordens de criação dos Actors e para receber os mapas. No caso da Visualization apenas será usado para receber mapas.

### *3.1.2 Maps*

No sentido de permitir efetuar simulações em ambientes realistas, usam-se mapas para representar as ruas e caminhos em que os atores (pedestres, automóveis, etc.) se podem movimentar. Para não haver a necessidade de ter de implementar um software para desenhar os próprios mapas, optou-se por usar um mapa com formato standard para gerar os mapas.

O objectivo principal será criar simulações com base em mapas de cidades reais. Será também importante que o carregamento desses mapas, por parte do simulador, seja rápido, fácil e também que não seja pesado para a simulação.

#### 3.1.2.1 Mapas Well-Known Text

Uma primeira abordagem para os mapas foi o formato *wkt* (well-known text). Este formato é usado pelo simulador *The One* e utiliza uma representação muito simples dos mapas reais sendo apenas constituído por pontos e linhas, o suficiente para gerar um mapa com as características básicas que são necessárias.

O formato *wkt* é um documento de texto que respeita algumas regras de escrita e permite criar pontos e linhas. Para fazer a distinção entre os dois tipos, é usada uma *label* no início de cada linha do documento. Usa-se a label *Point* para representação de apenas um ponto que é constituído pelas coordenadas *x* e *y*, ficando *POINT* (x y), sendo que, entre o *x* e o *y* apenas existe um espaço.

Para a representação de uma linha usa-se o *Linestring*, que é formado por um conjunto de pontos, ficando *LINESTRING* ($x_1$ $y_1$, $x_2$ $y_2$, ... $x_n$ $y_n$) . Uma *Linestring* pode corresponder a uma estrada, passeio ou caminho de ferro e o *POINT* representar uma paragem de autocarro, estacionamento ou ponto de interesse.

Um exemplo de um ponto e uma linha é apresentado a seguir:

```
POINT (17.7469493  10.1371223)
LINESTRING (-2.2511320754716966 -310.3971698113207, 29.331283 -285.582415,
188.37130188679242 7.6828679245283755)
```

Um par de coordenadas *x* e *y* pode constituir um *POINT* ou pertencer a vários *LINESTRING*. No caso de pertencer a mais do que uma *LINESTRING* significa que existe um cruzamento entre as linhas ou que uma linha nova começa a partir de uma já existente.

O ficheiro *wkt* pode ser editado manualmente mas se uma cidade tiver grandes dimensões (ou seja, centenas de linhas e pontos) é difícil construir o mapa. Por outro lado, ao editar diretamente o ficheiro *wkt* não se consegue ver como ele está a ficar, existindo uma maior probabilidade de erros na construção do mapa.

Por isso, recorreu-se a uma aplicação própria para criar mapas para o formato *wkt*, o *OpenJUMP*. O programa tem uma particularidade muito útil para a criação dos mapas, que é a possibilidade de abrir uma imagem e colocá-la no background do mapa que está a ser desenhado, tornando o mapa mais realista. É uma aplicação gratuita e multiplataforma que vai ao encontro dos princípios do nosso simulador open-source e multiplataforma, sendo assim mais fácil o uso das duas ferramentas na mesma máquina.

#### 3.1.2.2 Mapas OpenStreetMap

Mais tarde, quando se começou a implementação dos Actors, constatou-se que o formato *wkt* era limitado. Para se criar Actors mais realistas e com comportamentos mais específicos, sentiu-se a necessidade ter mais parâmetros associados aos pontos e linhas dos mapas, tais como, saber se uma linha é uma estrada, um caminho de ferro ou um passeio, quais os limites de velocidade de uma dada estrada, se é uma estrada principal ou secundária, se existe um ou dois sentidos numa dada via, etc..

Por isso, surgiu a necessidade de arranjar uma alternativa para a criação dos mapas. A solução arranjada foi o *OpenStreetMaps* (OSM). O OSM é um mapa mundial livre, e é feito e atualizado pela comunidade do OSM em todo o mundo, à qual qualquer pessoa poderá pertencer e pode ajudar a criar o mapa. É *open data*, toda a informação no mapa é editável e fornecida pelos utilizadores. O mapa pode ser visualizado no *OpenStreetMaps.org* e o *layout* é muito semelhante ao do *Google Maps*. Uma grande vantagem deste formato, para além de se ter acesso a toda a informação e ferramentas de trabalho gratuitas, é o facto de qualquer utilizador poder atualizar a qualquer momento o mapa, fazendo com que os mapas sejam atuais.

O OSM fornece aos utilizadores um software para edição dos mapas, denominado de *Java OpenStreetMaps Editor*. Este editor permite descarregar qualquer área existente do mapa para ser editado (*Figura 16*). Após o *download* da respetiva área, o software tem um interface onde se pode manusear o mapa (*Figura 17*). Depois de trabalhado, o mapa pode ser enviado para o servidor ou pode ser armazenado no disco do utilizador num ficheiro com o formato *osm*. Isto permite criar mapas para a simulação mais rapidamente do que com o formato *wkt*, no qual os mapas teriam de ser criados de raiz e manualmente.

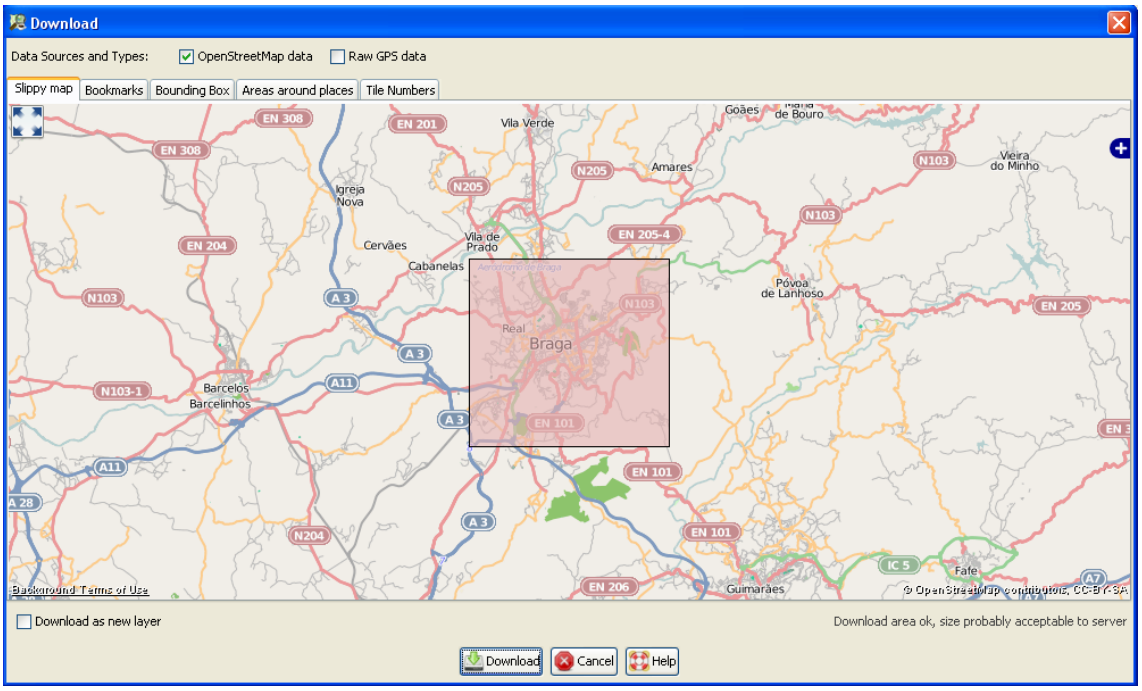


Figura 16 - Seleção da área que se pretende fazer o *download*.

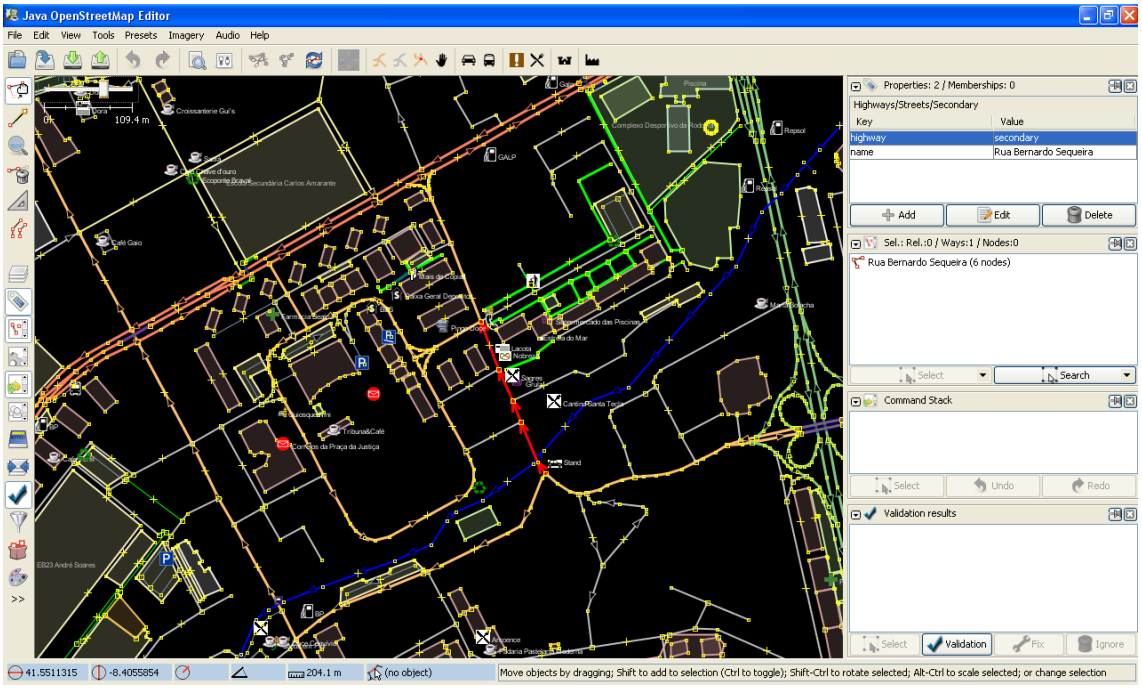


Figura 17 - Interface do *OpenStreetMap*.

A estrutura dos ficheiros OSM é ligeiramente diferente dos ficheiros *wkt*. Existem os tipos pontos e linhas como no *wkt*, as linhas (designadas por *way*) são um conjunto de pontos (designados por *nodes*). Inicialmente são criados os *nodes* que são constituídos por latitude e longitude, um identificador único para cada nó, um *timestamp* de quando foi criado e quem o criou, entre outras informações menos relevantes. A cada um destes nós podemos associar informação (designados por *tag*), como paragem de autocarro, local de estacionamento, cafés, lojas, etc.. Os *tags* dos *nodes* podem ser usados para criar pontos de interesse para os atores. De seguida, é possível ver um exemplo de alguns nodes:

```
<node id='1755055312' timestamp='2012-05-16T17:17:46Z' uid='11117' user='apires' visible='true' version='1' changeset='11615691' lat='41.514033' lon='-8.4500054'>
   <tag k='amenity' v='bus_station' />
 </node> -node com tags associadas
<node id='1755050296' timestamp='2012-05-16T17:10:17Z' uid='11117' user='apires' visible='true' version='1' changeset='11615691' lat='41.5335835' lon='-8.4382161' />
 <node id='1755050298' timestamp='2012-05-16T17:10:17Z' uid='11117' user='apires' visible='true' version='1' changeset='11615691' lat='41.524983' lon='-8.4475797' />
```

As *ways* são um conjunto de *nodes* colocados sequencialmente, formando uma linha que passa por todos os nós. Para reduzir o uso de informação, os *nodes* são apenas representados pelo seu identificador. As *ways* também podem ser acompanhadas por *tags* com informação do tipo de vias, como principais, secundárias ou residenciais; a quem se destina a via, como carros, comboios, pedestres; se a via tem um ou dois sentidos; a velocidade limite; etc.. É possível de seguida ver um exemplo de uma way com tags associadas:

```
<way id='22745048' timestamp='2012-04-20T13:25:53Z' uid='11117' user='apires' visible='true' version='5' changeset='11364074'>
   <nd ref='1755050296' />
   <nd ref='1755050298' />
   <tag k='area' v='yes' />
   <tag k='highway' v='pedestrian' />
   <tag k='name' v='Largo S. João do Souto' />
```

Estes mapas permitem-nos criar simulações de grandes cidades sem grande perda de tempo. Os atores podem ter mais especificidades graças as *tags* associados aos nós e as vias. A simulação poderá ser feita usando qualquer um dos mapas referidos, consoante as necessidades.

### *3.1.3 Actors e generators*

Os *Actors* têm um papel muito importante na simulação. Será a partir destas entidades que vai ser possível fazer os diversos estudos, investigações, testes, etc., ou seja, o que se pretende de um simulador de ambientes urbanos. Cada uma destas entidades vai representar uma entidade da vida real, por exemplo um *ActorPedestrian* representa um pedestre, um *Actor Car* representa um carro que está a ser conduzido por uma pessoa, etc.. Como se pretende que os *Actors* sejam o mais próximo dos atores reais, os mesmos serão dotados de comportamentos diferentes e terão de obedecer a determinadas regras. Os movimentos deles não serão aleatórios e terão de ter em conta os *Actors* que se deslocam ao seu redor.

Os *Generators* são as entidades responsáveis por criar novos atores. Para cada tipo de ator deverá existir, pelo menos, um gerador do tipo correspondente. Consoante a necessidade da simulação o Generator cria os diversos *Actors* com diferente cadência. Os geradores é que definem quais as propriedades iniciais dos atores criados como a sua posição inical, onde vão aparecer no mapa e qual a sua velocidade inicial. A cadência a que os atores são gerados, as suas posições iniciais e a sua velocidade são propriedades que podem variar de gerador para gerador, bastando apenas configurar, de modo a que vá ao encontro do que é pretendido para a simulação.

Na secção 4.1 será explicada com mais detalhe a relação entre os *Generators* e os *Actors*.

### *3.1.4 Reporting*

A função do *Reporting* é registar tudo o que acontece na simulação. Ao longo do tempo, esta entidade vai fazer o relatório de todas as atividades que ocorrem durante a

simulação. Esta função vai permitir que o utilizador, após a simulação, saiba tudo o que aconteceu e faça uma análise mais pormenorizada da simulação. Serão criados 3 ficheiros de *reporting* diferentes, um com informação das ações que ocorreram no simulador, desde ligações estabelecidades até erros que possam ter ocorrido, um outro ficheiro que vai guardando as coordenadas dos atores e um último ficheiro que vai armazenando a informação relativa à carga de cada *LocalCoordinator*.

O ficheiro com a informação das posições dos atores vai permitir que se volte a ver uma simulação anterior, bastando apenas executar o bloco *Visualization* (explicado na secção a seguir).

Todas as funções de *Reporting* encontram-se no *GlobalCoordinator*, pois é onde passa toda a informação do sistema.

### *3.1.5 Visualization*

O bloco *Visualization* não tem influência no decorrer da simulação, podendo a simulação decorrer sem este ser executado. No entanto, para os utilizadores terem uma noção de como a simulação está a decorrer e para uma melhor monitorização, o visualizador possuí um interface gráfico dos mapas que estão a ser usados e permite visualizar o comportamento dos atores (*Figura 18*). O visualizador será apresentado ao utilizador na forma de um *java graphics*. Tem ferramentas básicas para facilitar a visualização como o *Zoom in*, *Zoom out* e *Zoom total*. É possível arrastar o cenário de visualização e selecionar quais os atores pretendidos para aparecer no cenário, bastando selecionar no canto superior direito quais os atores que podem estar visíveis na visualização da simulação.

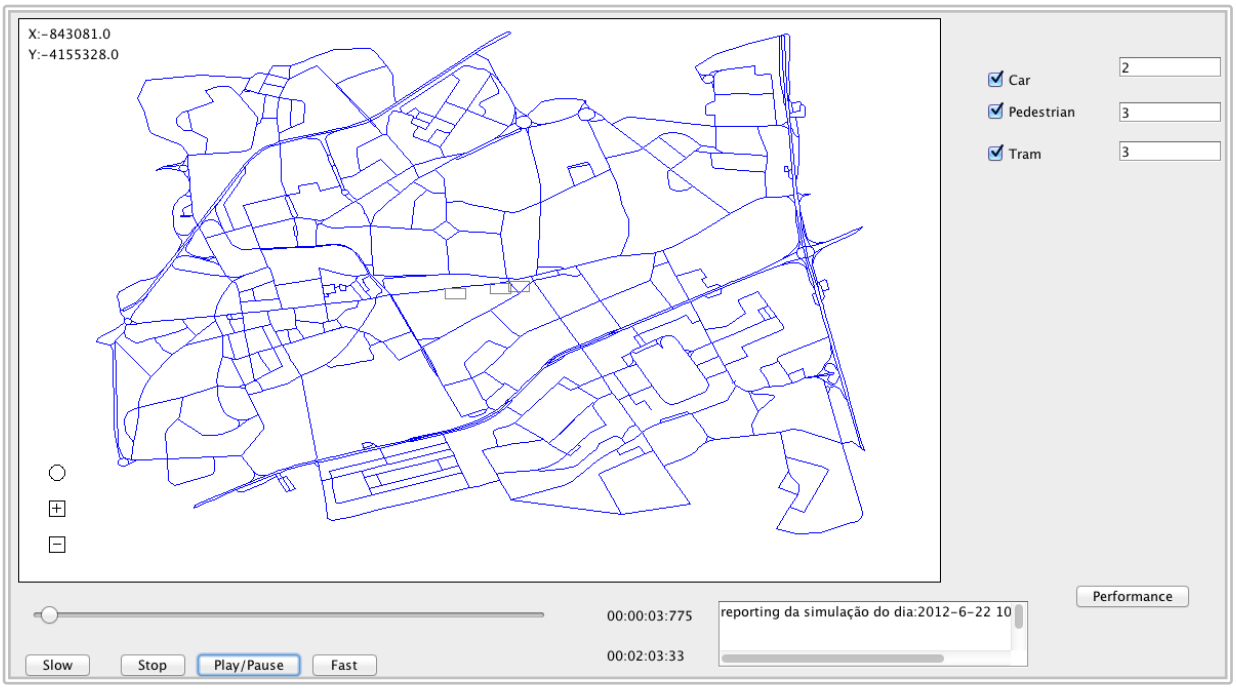


Figura 18 - Ambiente gráfico do visualizador do BartUM.

O visualizador permite observar as simulações em direto ou em diferido. Usando os ficheiros criados pelo módulo Reporting é possível fazer o "*play*" de uma simulação em diferido na *Visualization*.

A *Visualization* quando estabelece a primeira ligação com *GlobalCoordinator* recebe de imediato todos os mapas necessários para a simulação. Durante a simulação o visualizador vai recebendo atualizações do estado dos atores através do canal *Multicast* e, tal como os *LocalCoordinators*, também o visualizador inclui um *MulticastReceiver*.

# 4 Desenho e implementação dos atores

Neste capítulo serão apresentados os atores, o processo de desenho de cada um deles, e as suas respectivas implementações. O capítulo está divido em duas secções distintas. Na primeira secção é feita uma explicação do funcionamento e relação dos vários geradores e dos vários atores; na segunda parte é feita uma descrição de cada ator que foi pensado e como foi feita a sua implementação.

## 4.1 Atores e Geradores

Os *Generators* são inicializados pelo *GlobalCoordinator*. Cada *Generator* é uma thread, que funciona de forma independente. Os parâmetros iniciais do *Generator* são carregados a partir de um ficheiro de *properties* usado pelo *GlobalCoordinator* no início da sua execução, que tem as configurações iniciais da simulação. Os parâmetros que são carregados quando os *Generators* são criados são o tipo de ator que o *Generator* irá gerar, a probabilidade de geração de um determinado ator, as suas coordenadas iniciais x e y onde os *atores* vão aparecer no mapa, o nome dos mapas e das paragens que estão associados aquele tipo de *Actor*.

Segundo uma dada probabilidade, serão gerados os *Actors* com uma determinada cadência. Quando um *Actor* é gerado, é criada uma mensagem com os campos necessários para a criação do novo *Actor*, mensagem essa que é enviada para o *GlobalCoordinator*. Na mensagem vai também a velocidade inicial a que o ator começa. O valor da velocidade não é sempre o mesmo, para cada tipo de ator é definido um intervalo [velocidade $_{\text{mínima}}$, velocidade $_{\text{máxima}}$] e depois é calculado um valor aleatório dentro desse intervalo. Consoante o tipo de ator, o intervalo pode ter valores maiores ou menores. As mensagens enviadas pelos geradores podem variar de gerador para gerador, com mais ou menos parâmetros dependendo da necessidade de cada ator. De seguida, é possível ver um exemplo de um ficheiro *properties* e em destaque dois exemplos de geradores, gerador *Tram* e *Pedestrian*:

```
#Simulation Name
Global.Name=Simulação

#GlobalCoordinator IP address and port
GlobalCoordinator.IP=127.0.0.1
GlobalCoordinator.port=7575

#Multicast group and port
Multicast.IP=228.2.2.2
Multicast.port=7171

#Type of logging
Logging.Type=1

#Global Map of the simulation
Map.Number=1
Map.1=Braga.osm
Map.2=Paragens.wkt

#Genarator
Generator.Number=2
Generator.1=Tra1
Generator.2=Ped1

Tra1.Probability=0.05
Tra1.X=-843094.77
Tra1.Y=-4154704.139
Tra1.Maps=Map.1
Tra1.Stops=Map.2

Ped1.Probability=0.2
Ped1.X=-841906.64
Ped1.Y=-4154694.43
Ped1.Maps=Map.1
Ped1.Stops=Map.2
```

Após a receção da mensagem de criação do novo *Actor,* o *GlobalCoordinato*r, vai atribuir o *Actor* a um *LocalCoordinator*. De modo a balançar a carga por igual por todos os *LocalCoordinators* existentes, o *GlobalCoordinator* verifica qual das máquinas está menos sobrecarregada. Após essa análise, manda-se a mensagem que contém os dados do novo *Actor* pela rede, usando o protocolo TCP, para o endereço IP da máquina com menor carga.

Quando o *LocalCoordinator* recebe uma mensagem para criar um novo *Actor*, ele determina que tipo de *Actor* é que tem de ser criado e, de seguida cria-o passando todos os parâmetros que vêm na mensagem. No caso de ser a primeira vez a receber um determinado tipo de *Actor*, o *LocalCoordinator* vai requisitar ao *GlobalCoordinator* os

mapas que vêm identificados na mensagem do novo ator. Espera a receção dos mapas e só depois é que finaliza a criação do novo *Actor*. Assim como os *Generators*, os *Actors* serão threads que vão funcionar de forma independente do sistema.

Uma vez criado o novo *Actor,* é criado um *ActorStatus* para esse ator. O *ActorStatus* é um objecto criado na classe *ActorStatus*, com a informação necessária de um *Actor*, para passar para o resto do sistema. A informação que compõe o objecto *ActorStatus* é o ID do ator, as coordenadas atuais x e y e o seu vetor velocidade decomposto em velocidade em x ($V_x$) e velocidade em y ($V_y$), sendo estes últimos campos explicado mais a frente na secção 3.5 (Ator Tram).

Depois de criado o objeto *ActorStatus*, este é guardado na actorList. Esta lista serve para guardar a informação de todos os *Actors* da simulação, quer os *Actors* do próprio *LocalCoordinator* quer os dos outros *LocalCoordinators*. A actorList é uma tabela que na 1ªposição guarda o ID de cada *Actor* e na 2ª posição guarda o objecto *ActorStatus*. Esta lista apenas contém atores que se encontram ativos. Os atores podem ser eliminados da lista se forem eliminados ou por outra situação especial.

O ID de cada *Actor* é sempre diferente evitando sobreposição com outros atores. Sempre que ocorre uma atualização dos *Actors*, é atualizado o *ActorStatus* e de seguida é atualizado o estado na actorList. De modo a saber se um determinado ator existente na actorList pertence a um *LocalCoordinator*, foi criada uma lista local, a *LocalActors*, sempre que um ator é criado nesse *LocalCoordinator* é adicionado a essa lista.

Depois do ator criado, a *ActorList* vai sendo atualizada com os novos movimentos dos *Actors* e depois o *LocalCoordinator* envia essas mesmas atualizações para o *GlobalCoordinator* e para todos os outros LocalCoordinators.

De modo a poderem ser executados muitos *Actors* ao mesmo tempo, cada *Actor* é uma *thread* com especificações próprias consoante o tipo de ator que foi pedido.

## 4.2 Actor Generic

Nas secções seguintes serão descritos os vários atores que foram implementados. O simulador foi implementado na linguagem *Java* e, por isso, as descrições dos atores que

se seguem incluem requisitos, o modelo comportamental, os detalhes da sua implementação e a descrição dos geradores que lhes dão origem.

O *Actor Generic* não tem nenhuma função específica mas é o ator que proporciona a criação de todos os outros atores. Todos os atores serão uma extensão do *Actor Generic*, no entanto a diferença será que cada ator terá especificações e comportamentos apropriados para o tipo de entidade que se pretende simular.

### *4.2.1 Requisitos*

O *Actor Generic* deve comportar-se como um agente, atualizando o seu estado periodicamente e reportando o seu estado para o *SimStatus*.

### *4.2.2 Modelo comportamental*

O *Actor Generic* será apenas usado como uma base para criação de outros atores, por isso, não terá nenhum comportamento definido.

### *4.2.3 Implementação*

A classe *Actor* implementa o *Actor Generic* e é composta pelos métodos construtor da classe, pelo *run()* e pelo *moveActor()*. De seguida, é possível ver o código JAVA usado para implementar a classe do *Actor Generic*.

```
public class Actor extends Thread {
    String[] actorParams;
    String id = null;
    double x, y; //x and y represent the position of the actor
    int t = 100; //set update time pace (update rate = 1/t)
    double vx, vy, speed;
    double  lifetime; //lifetime of the actor
    boolean alive=true; // if actor is alive

    //Constructors of object
    public Actor(String actorDescription) {

        actorParams = actorDescription.split(":");
        id = actorParams[0];
        x = Double.parseDouble(actorParams[1]);
        y = Double.parseDouble(actorParams[2]);
        speed = Double.parseDouble(actorParams[3]);
    }

    @Override
    public void run()
    {
        //register the new actor as soon as it starts running
        SimStatus.setActorStatus(id, x, y,vx,vy);
        lifetime = 400000 + Math.random()*50000; //life time of the actor
        while (alive) //running while lifetime > 0
         {
            lifetime=lifetime - t;//decrements lifetime
            if(lifetime > 0)
            {
                 moveActor();//update the actor position
                 //publish the updated status
                 SimStatus.setActorStatus(id, x, y, vx, vy);
                 Thread.sleep(t);
            }
             else alive=false;

            }
       SimStatus.setActorStatus(id, alive);//status of actor changed to dead

    }


    //this is the method that models the actor's movement bevhaviour; in this case,
it does nothing
    Public void moveActor()
    {

    }
}
```

O construtor da classe Actor serve para inicializar o novo Actor com os parâmetros que foram passados pelo Generator, quando foi dada a ordem de criação do novo Ator. Os parâmetros são o ID do ator, a sua posição inicial em $x$ e $y$, a sua velocidade e o periodo de atualização.

O método *run* inicialmente regista o ator no *SimStatus*, passando para o Sim*Status* os parâmetros originais do ator como o ID, as suas coordenadas atuais *x* e *y*, e a sua velocidade decomposta em $V_x$ e $V_y$. Estes dois últimos parâmetros serão explicados mais à frente na secção 4.3 (*Actor Tram*).

De seguida, é calculado o tempo de vida do ator. O tempo de vida servirá para simular entidades que não serão mais relevantes para a simulação depois de uma dada ação, como acontece na vida real, quando as entidades saem da cidade e não voltam; que param indefinidamente (exemplo carros estacionados por muito tempo) ou entram em algum edifício. O tempo de vida será diferente para todos os atores, é definido um intervalo com [tempo de vida $_{mínimo}$, tempo de vida $_{máximo}$] e, de seguida, é gerado um valor aleatório dentro desse intervalo, que será o tempo de vida do ator.

Uma vez inicializado o ator, é criado um ciclo que está sempre a ser executado e sempre a cumprir uma sequência de ações. No ciclo, é atualizado o tempo de vida do ator, sempre que o ator atualiza a sua posição e é decrementado no tempo de vida o tempo de atualização. O tempo de atualização que está representado no código anterior pela variável *t*, representa o ritmo a que o ator atualiza o seu estado, este tempo foi definido em 100*ms*. O tempo é fixo e igual para todos os atores. Quando o tempo de vida acaba, o ator "morre" e é alterado o seu estado de ativo para inativo no *SimStatus*. A *thread* do ator após este ficar inativo deixa de ser executada e o *Actor* é eliminado do A*ctorsList*.

Após ser atualizado o tempo de vida é evocado o método *moveActor*. O método *moveActor* será a estrutura central de cada ator, onde são invocados os métodos que caraterizam o comportamento do ator em termos de movimento. Neste caso, em específico, não tem nenhum comportamento. Depois de ter executado o método *moveActor()*, é atualizado o estado atual do ator no *SimStatus* e, de seguida, introduz-se uma pausa que faz com que não exista atividade do ator durante um tempo fixo, o periodo de atualização (*t*). Este ciclo é apenas interrompido se o ator "morrer" na simulação. A seguir, apresenta-se o funcionamento geral do *Actor Generic* (*figura 19*).

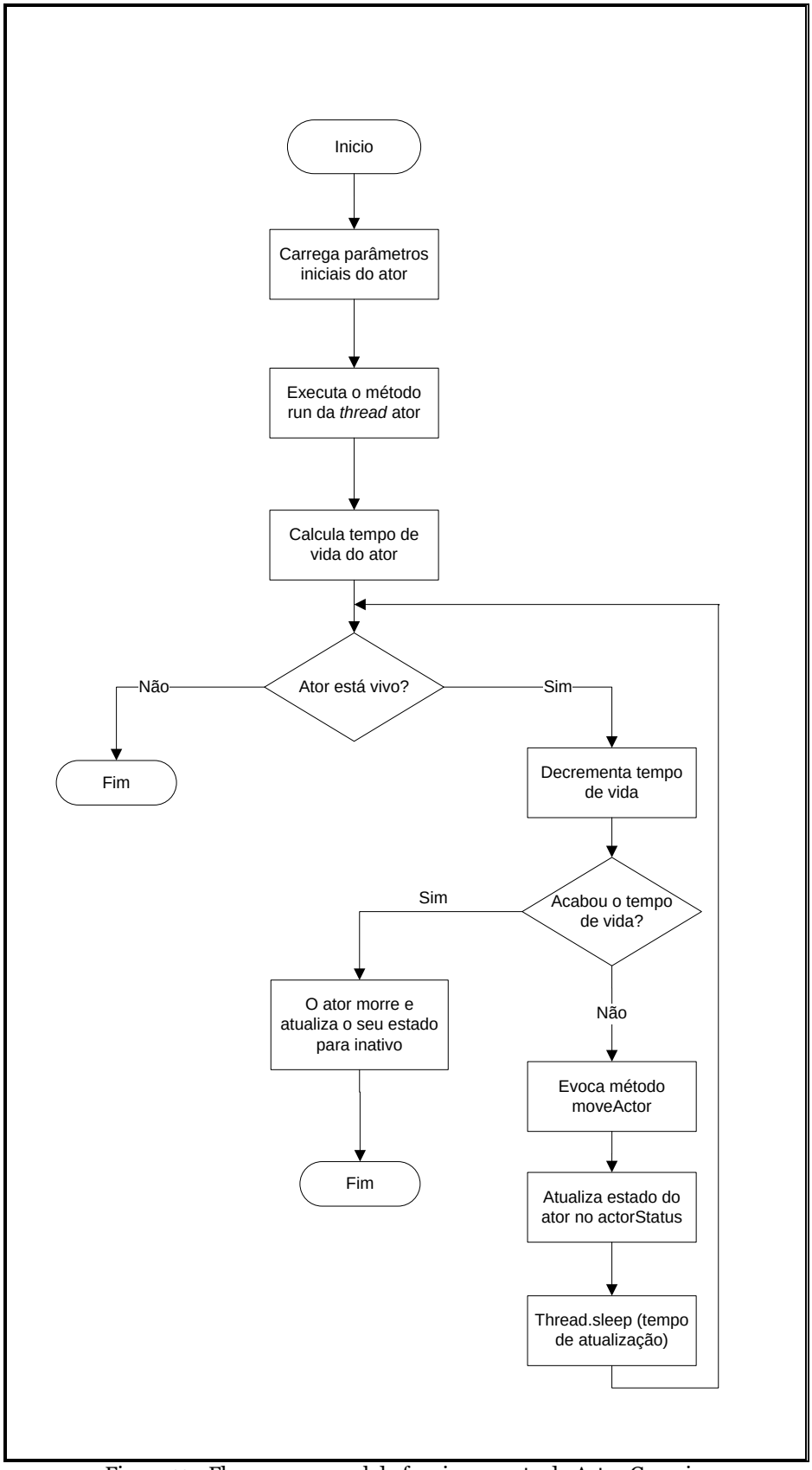


Figura 19 - Fluxograma geral do funcionamento do Actor Generic.

### *4.2.4 Generator*

Assim como o *Actor Generic*, o seu gerador tem uma função semelhante. Nunca é usado diretamente, todos os geradores são uma réplica deste gerador, mas com especificações próprias do tipo de ator que têm de gerar. De seguida, é possível ver o código *Java* usado para implementar a classe do *Generator* do *Actor Generic*.

```
public class Generator extends Thread {
    String genName; //Generator name
    double p; //generation probability
    double xi, yi; //initial position of the actor
    String actorID;
    int count = 0;
    int genPeriod = 2000; //the time interval between iterations of the generator

    //Constructors of object
    public Generator(Properties prop, String genName)
    {
        this.genName = genName;
        //read the generator parameters from the configuration file
        p = Double.parseDouble(prop.getProperty((genName + ".Probability")));
        xi = Double.parseDouble(prop.getProperty((genName + ".X")));
        yi = Double.parseDouble(prop.getProperty((genName + ".Y")));
    }

    //Thread's run method
    @Override
    public void run()
    {
        while(true) {
            if(newActorQ()) {
                actorID = genName + "." + count;

                String newActorDescription = actorID + ":" + xi + ":" + yi;
                GlobalCoordinator.createNewActor(newActorDescription);
                count++;
            }
            try {
                Thread.sleep(genPeriod);
            } catch (InterruptedException ex) {
            Logger.getLogger(Generator.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }
    //this methods decides if a new actor is to be created: returns true if the
random number is higher than p
    public boolean newActorQ()
    {
        double value = Math.random();
        if(value > p)
            return true;
        else
            return false;
    }
}
```

A classe *Generator* foi implementada como sendo uma *thread*, o que significa que tem um funcionamento independente do que passa no sistema. As *Generators* são iniciadas pelo *GlobalCoordinator* e os parâmetros são carregados a partir do respetivo ficheiro *properties*. Esses parâmetros são guardados na classe.

A classe *Generator* apenas possui três métodos, o construtor da classe *Generator,* o *run* e o *new_actor,* sendo que a *new_actor* está a ser constantemente evocada pela função *run*.

O método *Generator* serve apenas para carregar os parâmetros que são recebidos do ficheiro de *properties*, as coordenadas iniciais do ator e probabilidade de geração de um ator.

O método *run* implementa tudo o que a *thread* tem que fazer como processo independente. Uma vez iniciado este método, a *thread* é iniciada, é feita a execução método e depois deixa de existir. No método é evocado a função *new_actor* e, de seguida, é testado o que foi devolvido por essa função e se for *true,* cria-se um novo *Actor*. Para não se gerar apenas um ator, é criado um ciclo em que a sua condição é sempre verdadeira, dentro do método *run*, ou seja, ficando a fazer este ciclo até ao fim da simulação. Para evitar que a geração dos atores seja demasiado grande, a *thread* é "adormecida" durante um período de tempo em cada ciclo. O tempo de repouso será diferente para cada tipo de ator.

O método *new_actor* apenas é evocado com o objectivo de testar se será criado um ator ou não. É gerado um valor aleatório, entre 0 e 1 e é verificado se o valor gerado está dentro do limite definido para gerar um ator. Em função do resultado, o método devolve um *true* se for para criar um ator, ou seja, se o valor estiver dentro do intervalo, ou devolve um *false,* se o valor estiver fora do intervalo.

## 4.3 Ator Random

O *Actor Random* é o ator mais simples de todos, foi pensado para imitar apenas movimentos erráticos. Este ator é baseado no modelo *Random Walk* e os movimentos que o ator faz, são completamente aleatórios.

### *4.3.1 Requisitos*

O *Actor Random* deve comportar-se segundo o modelo *Random Walk*, efetuando movimentos aleatórios, atualizando o seu estado periodicamente e reportando o seu estado para o *SimStatus*.

### *4.3.2 Modelo comportamental*

Com base nos requisitos do ator, foi desenhado um modelo comportamental. Como o *Actor Random* tem um comportamento aleatório, as coordenadas para onde o ator se vai deslocar são geradas aleatoriamente. Inicialmente, o ator é colocado num ponto, dentro da área de simulação, depois gera um novo destino e desloca-se até ele com uma velocidade constante. Uma vez chegado ao destino, volta a gerar um novo destino e repete o processo sucessivamente. Como não existem limites na área de simulação, o ator não tem qualquer restrição espacial. Como o ator normalmente está sempre a circular nas cidades, será definido um tempo de vida muito elevado de modo a ficar sempre em movimento durante a simulação.

### *4.3.3 Implementação*

A classe *ActorRandom* é uma extensão da classe *Actor*, assim sendo, tem uma estrutura igual à da classe *Actor*. A diferença neste caso, está no método *moveActor(),* que no caso do ator genérico não tinha nenhum comportamento. No método *moveActor é* definido o comportamento do *Actor Random*.

A cada atualização do seu estudo, a posição é calculada usando as seguintes expressões:

$$X = x + (r_1 * 100 - 50) * v \quad \text{(Eq. 1)}$$

$$Y = y + (r_2 * 100 - 50) * v \quad \text{(Eq. 2)}$$

em que:

X - coordenada em x da nova posição

Y - coordenada em y da nova posição

x - coordenada atual em x

y - coordenada atual em y

$r_1$ - variável que terá um valor aleatório no intervalo [0,1]

$r_2$ - variável que terá um valor aleatório no intervalo [0,1]

v -valor da velocidade a que o ator se desloca

Os valores de x e y iniciais e a velocidade são definidos no momento da criação do ator, pelo *Generator Random*. A velocidade é um valor compreendido entre [velocidade mínima, velocidade máxima]. No resto da simulação o valor da velocidade não se altera, enquanto que o x e y serão alterados a cada atualização do ator.

O *Actor Random* fica a ser executado até o tempo de vida do ator acabar.

### *4.3.4 Generator Random*

O *Generator Random* cria atores com uma certa cadência, na posição inicial (*x,y)*, e com uma dada velocidade. Os valores da probabilidade de geração de atores e a sua posição inicial estão definidos no ficheiro de configuração inicial. A velocidade a que o ator começa a simulação não é sempre a mesma, está contido num intervalo compreendido entre [velocidade mínima, velocidade máxima]. A variação da velocidade dentro do intervalo é aleatória. Os valores de velocidade mínima e máxima apenas podem ser alterados no código do gerador.

## 4.4 Ator Tram

O *tram* é um transporte que se desloca sobre carris, ou seja, o percurso apenas pode ser feito onde existirem os carris, o que faz o seu movimento ser muito restrito. Em cada carril podem existir vários *trams*, mas nunca nenhum *tram* pode ultrapassar outro. Normalmente uma cidade não tem muitos *trams* em funcionamento ao mesmo tempo, ou seja, o número de *Actors Tram* será reduzido em comparação com os outros atores, nomeadamente carros e pedestres.

### *4.4.1 Requisitos*

O *Actor Tram* deverá ter um comportamento baseado nos modelos *City Section Mobility Mode* e *Manhattan Mobility Model*. O ator terá de se deslocar sobre mapas pré definidos e, sempre que chegar a um cruzamento, escolher por qual caminho deve seguir. O ator tem de ter em conta todos os outros atores que o rodeiam, tomando decisões como acelerar e abrandar, tem de atualizar o seu estado periodicamente e reportar o seu estado para o *SimStatus*.

### *4.4.2 Modelo Comportamental*

A pensar nestas especificações foi idealizado o comportamento para o *Actor Tram*. O ator tem obrigatoriamente de se deslocar sobre as linhas pré definidas, não podendo ultrapassar um outro ator que se desloque à sua frente e sendo obrigado a abrandar, ou mesmo parar, para não chocar com outro ator que siga à sua frente. No caso de existir uma interseção de carris e dois trams forem em rota de colisão, eles têm de diminuir ou aumentar a sua velocidade de maneira a evitar a colisão. Como normalmente os trams estão sempre em movimento numa cidade e não se deslocam para fora da cidade, o seu tempo de vida será infinito, mantendo assim os trams sempre ativos na simulação.

### *4.4.3 Implementação*

Depois de pensados os requisitos e o modelo comportamental, foi estruturada uma abordagem de modo a conseguir respeitar essas caraterísticas.

Quando o *Actor Tram* é criado, são carregados todos os parâmetros e, como acontece nos atores anteriores, a sua posição e velocidade inicial e o nome dos mapas necessários para se deslocar. Os atores descritos anteriormente não tinham a necessidade de se deslocar sobre mapas, visto que eram atores mais simples, mas neste caso é necessário a utilização de mapas pois, na realidade, os trams não têm comportamentos aleatórios e precisam de obedecer a caminhos específicos para a sua deslocação.

Como este ator vai ter um comportamento muito mais específico que o *Actor Random*, é necessário ter algumas precauções extra ao colocar o ator na simulação. O facto do *Actor Tram* ter de se deslocar sobre linhas do mapa, obriga a que seja necessário colocar inicialmente o ator numa dessas linhas. Através das coordenadas iniciais do ator pode-se procurar um ponto no mapa que tenha exatamente as mesmas coordenadas do ator. Preferencialmente as coordenadas iniciais do ator são dadas tendo em conta um ponto existente no mapa, para facilitar a procura do ponto onde o ator vai ser colocado. Se eventualmente as coordenadas iniciais não existirem no mapa, o ator não conseguirá começar a deslocar no mapa, na altura da escrita da dissertação já estava a ser pensada numa solução mas que não ficou pronta a tempo para ser documentada.

Uma vez colocado o Ator num ponto existente no mapa, é necessário procurar quais os possíveis caminhos que o ator poderá seguir. Os mapas são compostos por pontos, que depois são ligados pela ordem com que se encontram no ficheiro do mapa, formando linhas contínuas que serão as representações das estradas, passeios ou carris. Foi criado um método *findNextDestination*, para fazer a procura dos pontos para onde o ator se pode deslocar.

#### *findNextDestination*

A ligação entre dois pontos vai criar um segmento de reta e será nestes segmentos que os atores se irão deslocar. Um segmento tem sempre coordenadas iniciais e finais.

Sempre que o ator precisa de saber um novo destino, ele estará no início de um segmento à procura do fim de segmento que será o novo destino.

Como foi explicado na secção 3.1.2, os mapas são constituídos por linhas e estas linhas são formadas pelas ligações feitas entre os pontos existentes. De forma a guardar a informação dos mapas contida nos ficheiros OSM e *wkt*, foi criado um *parser* que lê todos os pontos e linhas que existem nos ficheiros e guarda a informação numa instância da classe Maps.

Os dados são guardados em objetos que vão conter as linhas existentes e os pontos que cada linha tem. A estrutura *Maps* foi pensada para guardar os mapas de modo a tornar toda a informação acessível e organizada, foi dividido em 4 partes, cada uma com uma função específica. Foram desenvolvidas 4 classes para guardar a informação dos mapas, a classe *Global_Map*, a *Map_line*, a *Map_Point* e a *Point_Line*, e cada uma destas classes cria um objecto com os dados relacionados com a sua função.

O *Global_Map* é usado para guardar a informação relativa aos mapas que são usados na simulação. É no construtor desta classe que é efetuado o carregamento da informação contida nos ficheiros dos mapas. Por cada ficheiro de mapas que tenha que ser carregado, é criado um novo objeto *Global_Map*. O *Global_Map* é composto por duas listas, a *line_list* que serve para guardar todas as linhas que existem no ficheiro e é composta por objetos do tipo *Map_Line* (explicado a seguir); e a *point_list* que guarda todos os pontos independentes do mapa, e é composta por objetos do tipo *Map_Point* (explicado mais a frente).

O *Map_Line* serve para guardar as linhas existentes no ficheiro do mapa. É composto por duas variáveis, uma que serve para identificar cada linha do mapa existente na *line_id*, e a outra que vai armazenar todos os pontos existentes em cada linha a *point_line*.

O *Map_Point* terá uma função semelhante ao *Map_Line*, mas terá de armazenar os pontos individuais que existam no ficheiro do mapa. Para permitir distinguir os vários pontos, será atribuído um identificador a cada ponto, o *point_id*. Será composto

por mais duas variáveis o *point_x* e *point y* para guardar as coordenadas *x* e *y* do ponto respetivamente.

Por último, foi criado o *Point_Line* para se armazenar a informação relativa aos pontos que constituem cada linha do mapa. Para ser possível saber a que linha cada ponto pertence, foi criado um identificador de linha para os pontos, a variável *line_id* e terá o mesmo valor da variável com esse nome da classe *Map_Line*. Como este objeto foi criado para guardar a informações dos pontos, tem de guardar nas variáveis *point_x* e *point_y* as respetivas coordenadas x e y.

Inicialmente o ator é colocado num ponto do mapa, esse ponto será o início do segmento (ponto P1 da *Figura 20*). O próximo passo será descobrir qual é o novo destino, ou seja, descobrir qual o próximo ponto que o ator se pode deslocar, no exemplo das figuras 21 e 22 os destinos são o P1, P2, P3 e P4. Para fazer pesquisa do novo destino, basta encontrar o ponto atual (ponto P1 com as coordenadas (0,0)) do ator no *hashmap,* e o valor que vem imediatamente a seguir as coordenadas do P1, qu representam a outra extremidade do segmento (ponto P2 com as coordenadas (100,0)) que o ator se pode deslocar. Um ponto pode ter um ou vários destinos possíveis e para se conseguir determinar se existe mais do que uma opção de trajeto, sempre que se procura um novo destino, será feita uma pesquisa por todos os possíveis destinos. Para se conseguir procurar todos os possíveis pontos é preciso percorrer o hashmap do início ao fim e sempre que aparecer as coordenadas do ponto P1 é guardado o par de coordenadas imediatamente a seguir. Se existir no final da pesquisa mais do que um ponto é escolhido um deles aleatoriamente, com igual probabilidade de escolha para qualquer ponto encontrado.

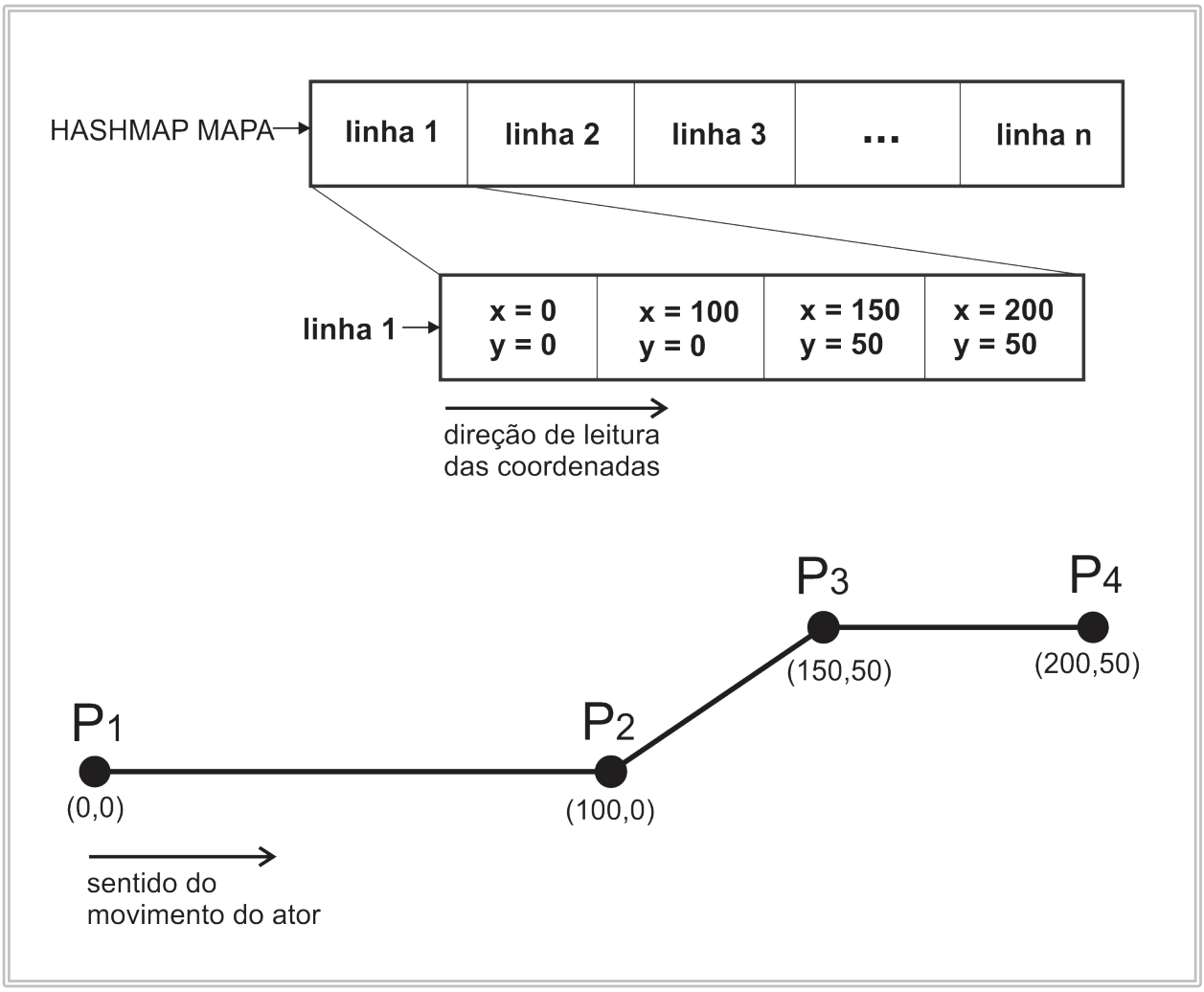


Figura 20 – Procura de nova coordenada

Quando um ator chega ao fim de uma linha que não tem continuação, é invertido o método de pesquisa, ou seja, é procurado o ponto do fim para o início do *hashmap* (exemplo da *Figura 21*). Em vez de o destino ser o ponto imediatamente a seguir ao ponto atual do ator, será o ponto anterior. Isto é feito de modo a que o ator nunca pare por não haver caminho para percorrer.

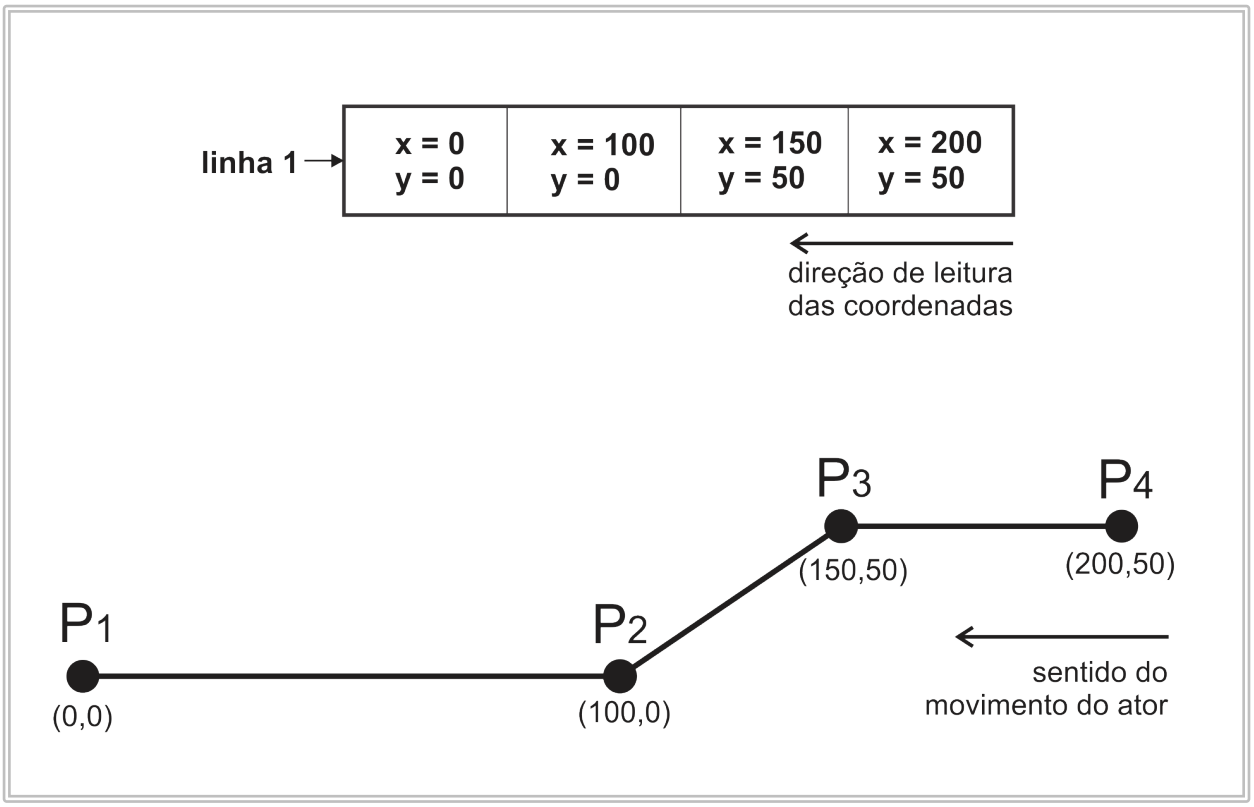


Figura 21- Procura de nova coordenada no sentido inverso

O método *findNextDestination()* é sempre evocado quando o ator chega a um destino e precisa de um novo destino.

De modo a que o ator saiba em que direção e sentido se deve dirigir quando está a movimentar-se e para facilitar a interação com os outros atores, é criado um vetor velocidade que carateriza o ator. O vetor é criado e atualizado no método *setSpeedVector*. Através deste vetor velocidade, será possível tomar decisões sobre o movimento do ator consoante o movimento dos outros atores e vice-versa, por exemplo evitar colisões entre atores, através do conhecimento do sentido e direção que eles se deslocam. O vetor é criado e atualizado no método *setSpeedVector*.

***setSpeedVector***

A direção do vetor será criada de modo a ser paralelo ao segmento em que o ator se está a deslocar. O sentido do vetor tem o sentido, do início do segmento para o fim do segmento. A norma do vetor é a velocidade a que o ator se está a deslocar e o vetor tem início no ponto onde o ator se encontra. Como se está a trabalhar com sistema de coordenadas $x$ e $y$, o vetor velocidade é divido em vetor em $x$ e vetor em $y$. Calculando a norma destes dois vetores ($V_x$ e $V_y$) temos o valor da velocidade. Uma vez sabendo qual o segmento em que o ator se está a deslocar e a velocidade a que o ator se desloca é possível usar as seguintes fórmulas para calcular o vetor.

$$V_x = \frac{V * (xfs - xis)}{\sqrt{(xfs - xis)^2 + (yfs - yis)^2}} \quad \text{(Eq. 3)}$$

$$V_y = \frac{V * (yfs - yis)}{\sqrt{(xfs - xis)^2 + (yfs - yis)^2}} \quad \text{(Eq. 4)}$$

em que:

$x_{is}$ - variável com o valor do início do segmento em x

$x_{fs}$ - variável com o valor do fim do segmento em x

$y_{is}$ - variável com o valor do início do segmento em y

$y_{fs}$ - variável com o valor do fim do segmento em y

$v_x$ - valor do vetor velocidade em x

$v_y$ - valor do vetor velocidade em y

v - velocidade atual do Actor

Após as configurações iniciais do *Actor Tram* é criado o ator no *SimStatus*. Depois começa o ciclo que existe no método *run()* do *Actor Generic*, pois o *Actor Tram* é uma extensão do *Actor Generic*, ou seja, chama o método *moveActor()* e atualiza a

posição no *ActorStatus* e faz uma pausa e recomeça. De seguida, será explicado com mais detalhe o método *moveActor* do *Actor Tram*.

***moveActor***

No fluxograma a seguir, pode-se ver a estrutura geral da abordagem usada para implementar o comportamento do *Actor Tram*.

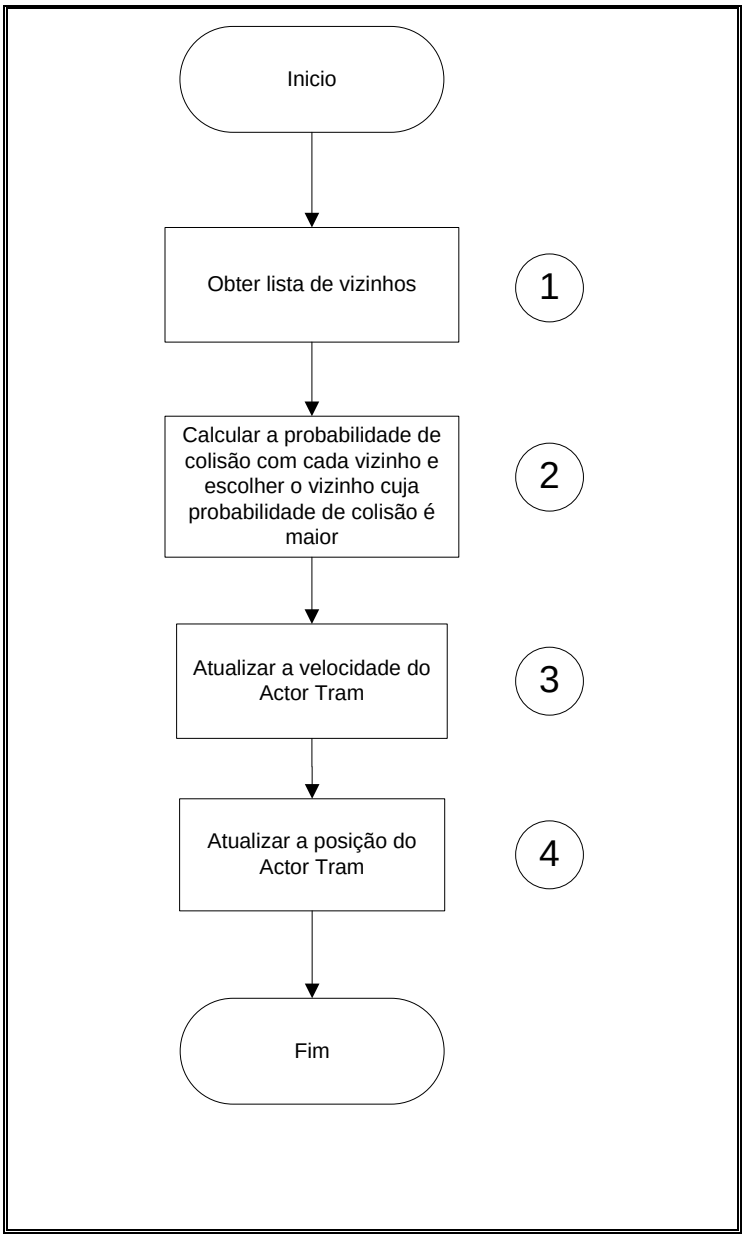


Figura 22 – Fluxograma geral do Actor Tram

Neste fluxograma da *Figura 22* os blocos foram numerados de forma a ser mais fácil a identificação de cada bloco.

No bloco 1 é onde o ator procura atores vizinhos, ou seja, atores dentro de uma área definida à sua volta e que podem representar uma possível colisão com o ator. Este bloco é implementado na forma de um método *getNeighbours()*.

No bloco 2 é calculada a probabilidade de colisão (*probCol*) com algum vizinho que possa existir. A *probCol* é uma variável que atua diretamente na velocidade do ator. Se *probCol*> *0*, a diminuição da velocidade será proporcional ao valor da *probCol*, ou seja, quanto maior for a *probCol* menor será a velocidade do ator.

No bloco 3 é atualizada a velocidade do ator, poderá ser uma redução ou um aumento de velocidade, dependendo do valor da probabilidade de colisão estimada anteriormente.

Por fim, no bloco 4 são atualizadas as coordenadas do ator, dependendo da velocidade a que se desloca o ator.

Com o fluxograma seguinte, pretende-se retratar com mais detalhe o fluxograma apresentado anteriormente para um melhor entendimento da estrutura do método *moveActor (Figura 23)*.

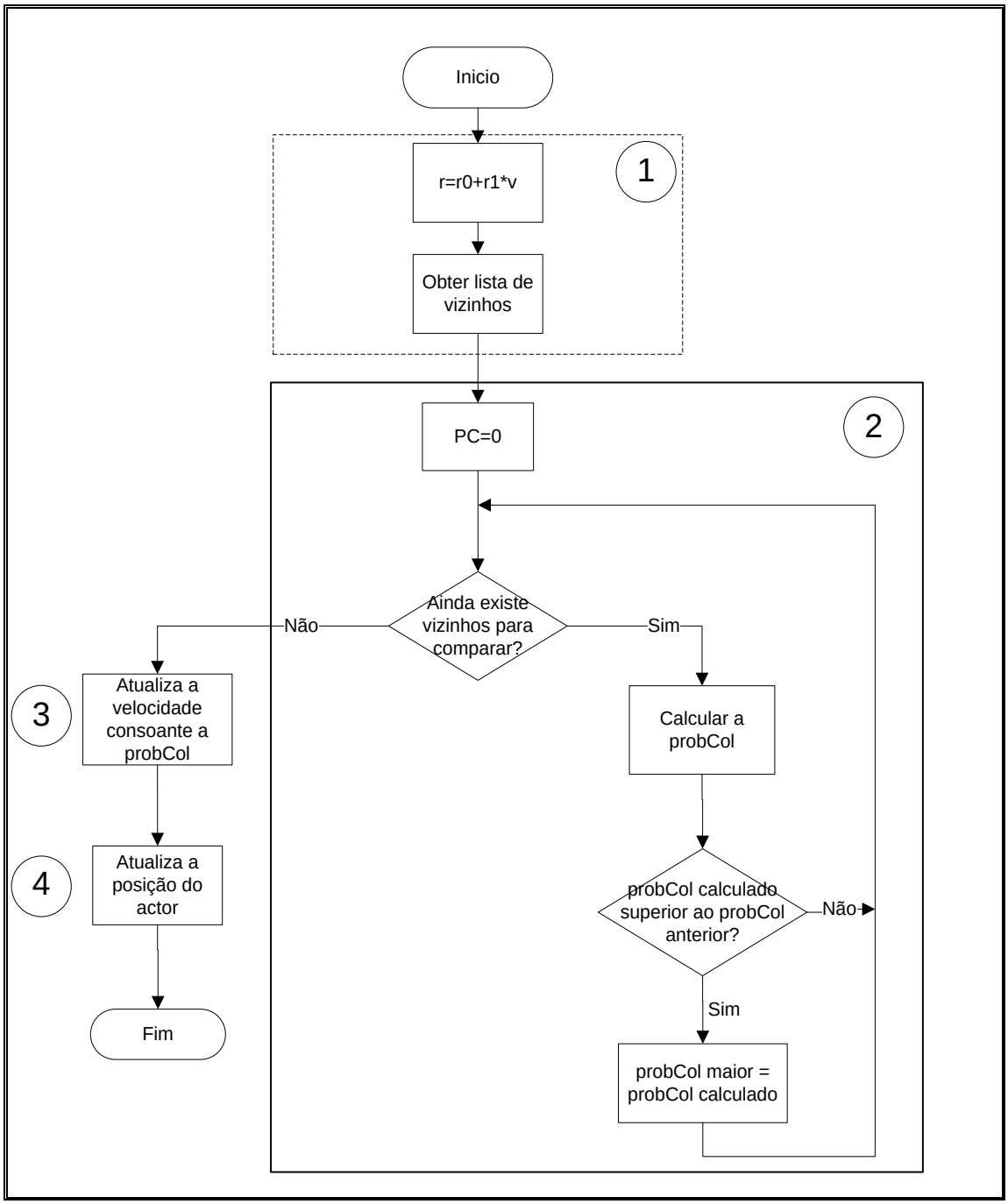


Figura 23 – Fluxograma do método moveActor() do Actor Tram

De seguida, será feita uma explicação mais detalhada do fluxograma apresentado. Inicialmente, o ator vai procurar atores vizinhos que possam originar uma colisão. A zona de procura de vizinho depende de um raio previamente calculado. De modo a que o raio seja adequado ao tipo de ator que está a ser executado, por exemplo, um *Actor Car* que se desloca a uma grande velocidade, tem de ser alertado dos vizinhos,

atempadamente, de modo a conseguir prevenir uma colisão, ao contrário do *Actor Tram* que se desloca muito devagar, tem maior tempo de reação antes de uma colisão. Assim, o raio é dependente da velocidade atual a que o ator se desloca, sendo calculado através da seguinte expressão:

$$r = r_0 + r_1 * v \quad \text{(Eq. 5)}$$

em que:

r - raio de procura de vizinhos.

r0 - representa a distância mínima da procura de vizinhos. Este valor é pré definido.

r1 - representa a distância adicional da procura de vizinhos. Este valor é pré definido.

v - velocidade a que o ator se desloca.

Após ter sido calculado qual a área de procura de vizinhos, é usado o método *getNeighbours* para fazer a pesquisa pelos atores que se encontram dentro da área de procura.

**getNeighbours**

Como o objetivo deste método é procurar atores vizinhos de modo a evitar colisões, a cada atualização do movimento do ator, o método é evocado para assim o ator se poder adaptar às alterações dos outros atores. Para este método são passados os parâmetros ID, as coordenadas atuais do ator e o raio que se calculou anteriormente. Espera-se que seja retornado, se existirem vizinhos, um *HashMap* com todos os vizinhos encontrados.

O método *getNeighbours* tem uma estrutura como se pode ver na *Figura 24*:

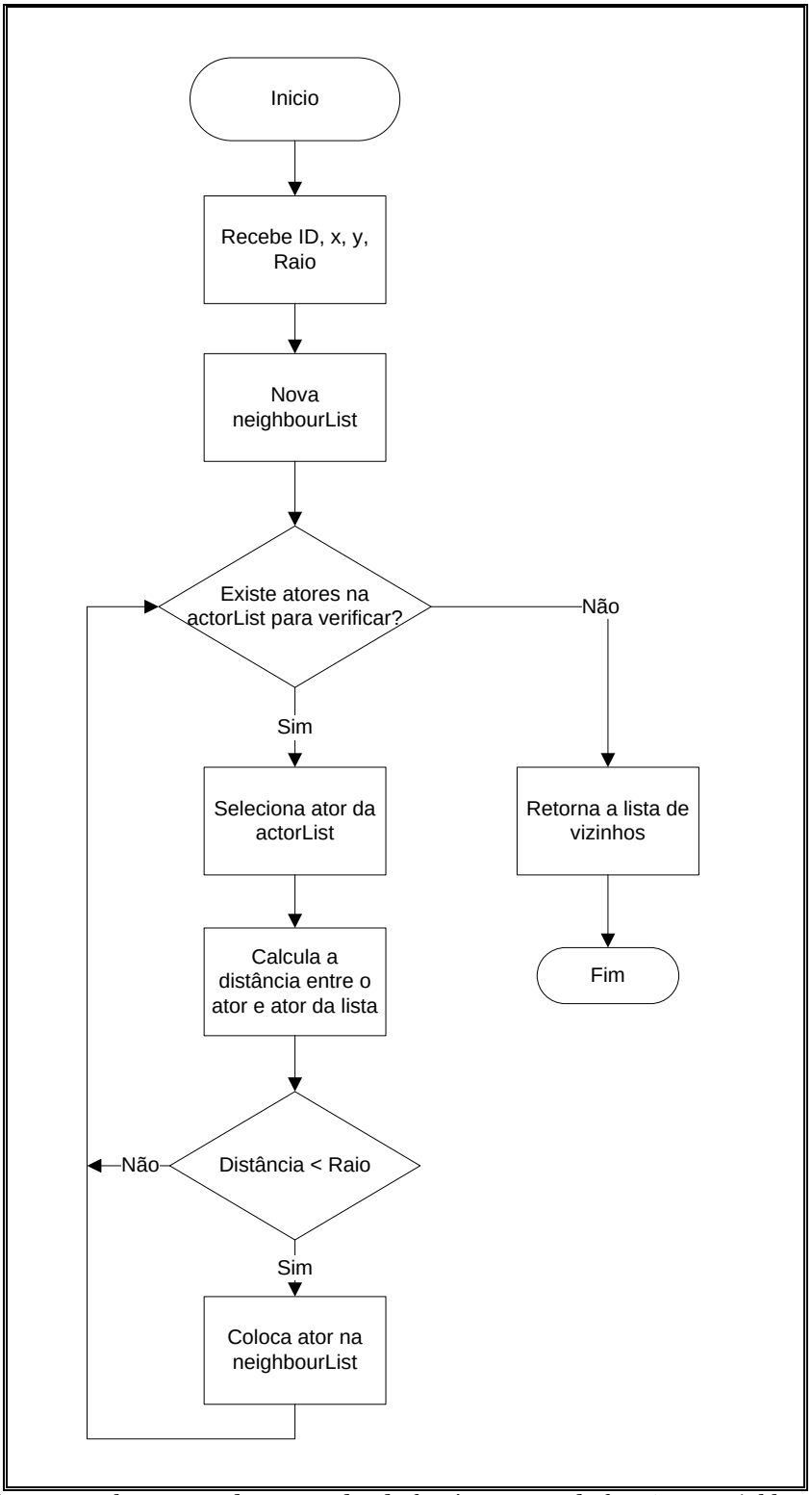


Figura 24 - Fluxograma demonstrador do funcionamento da função *getNeighbours*

Como se pode ver neste fluxograma, após a receção dos parâmetros passados pelo ator, é criada um *HashMap neighbourList*, com características muito semelhantes à

*actorList,* que é a lista onde se encontram todos os atores da simulação. Na *neighbourList* são colocados todos os atores que sejam vizinhos. A informação que irá para a lista será o ID do ator vizinho e o respectivo *ActorStatus*.

Se existirem atores na *ActorList*, são calculadas as distâncias a que se encontram os atores do Actor que evocou o método. Se essa distância for menor que o valor do raio, esse atores são considerados vizinhos e, são guardados na *neighbourList*, caso contrário, passa para o próximo ator da lista. Uma vez percorrendo a *actorList* até ao fim, é devolvido para o ator a *neighbourList*.

A fórmula usada para calcular a distância entre dois atores, é a distância euclidiana, que se pode ver de seguida:

$$d = \sqrt[2]{(x_{ator} - x_{ator\ viz.})^2 + (y_{ator} - y_{ator\ viz.})^2} \quad \text{(Eq. 6)}$$

em que

d - distância entre dois atores
$x_{ator}$ - coordenada em x do Actor
$y_{ator}$ - coordenada em y do Actor
$x_{ator\ viz.}$ - coordenada em x do possível ator vizinho
$y_{ator\ viz.}$ - coordenada em y do possível ator vizinho

Para uma melhor perceção do funcionamento do *getNeighbours*, na *Figura 25* é elaborado um exemplo de como funcionaria, no caso de um ator pedir a lista de vizinhos:

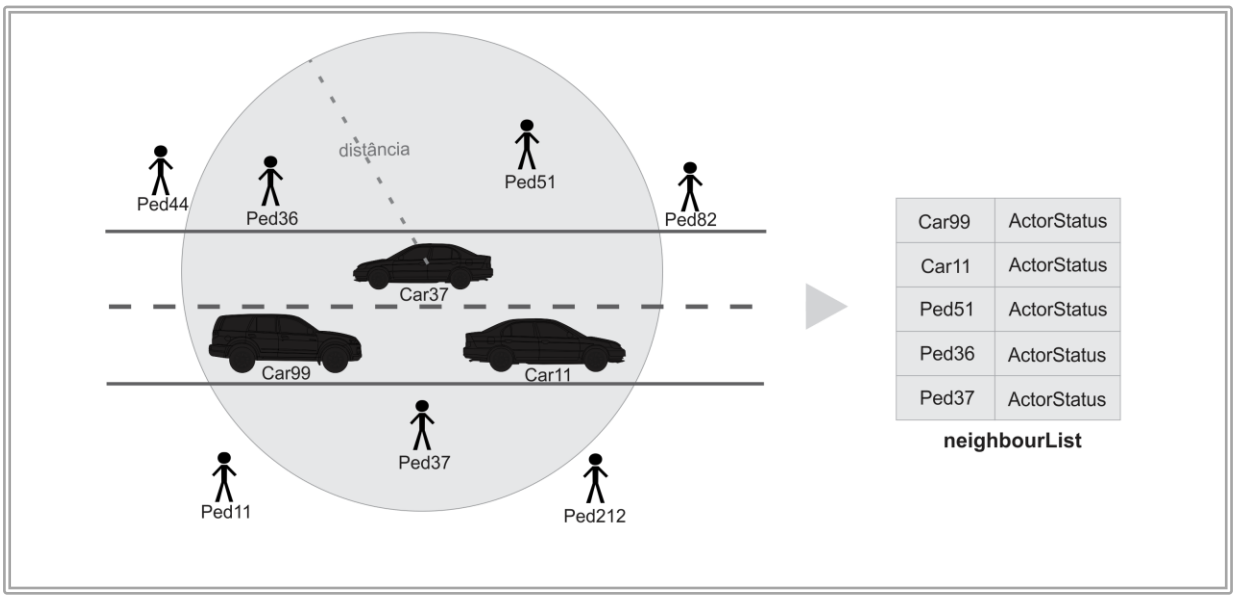


Figura 25 - Exemplo de procura de vizinhos do Car 37 e a respetiva neighbourList.

Após ser retornada a lista de vizinhos ao ator, se não existirem vizinhos na sua área de procura, o ator apenas precisa de atualizar a sua posição para uma nova coordenada. Se existirem, o ator tem de determinar se existe uma possível colisão com os atores vizinhos. Para determinar qual a probabilidade de colisão com um vizinho foi criado o método *collisionProbability*.

**collisionProbability**

O ator calcula a probabilidade de colisão (*probCol*) com os vizinhos que existirem. Este método calcula a *probCol* com cada um dos vizinhos, num intervalo entre 0 e 1. Após o cálculo, compara-se sempre se a *probCol* que se calculou é superior ao valor da *probCol* calculada no vizinho anterior. O método retorna o maior valor de todas as *probCol* calculadas.

No fluxograma a seguir, pode ver-se o funcionamento geral do método *collisionProbability (Figura 26)*:

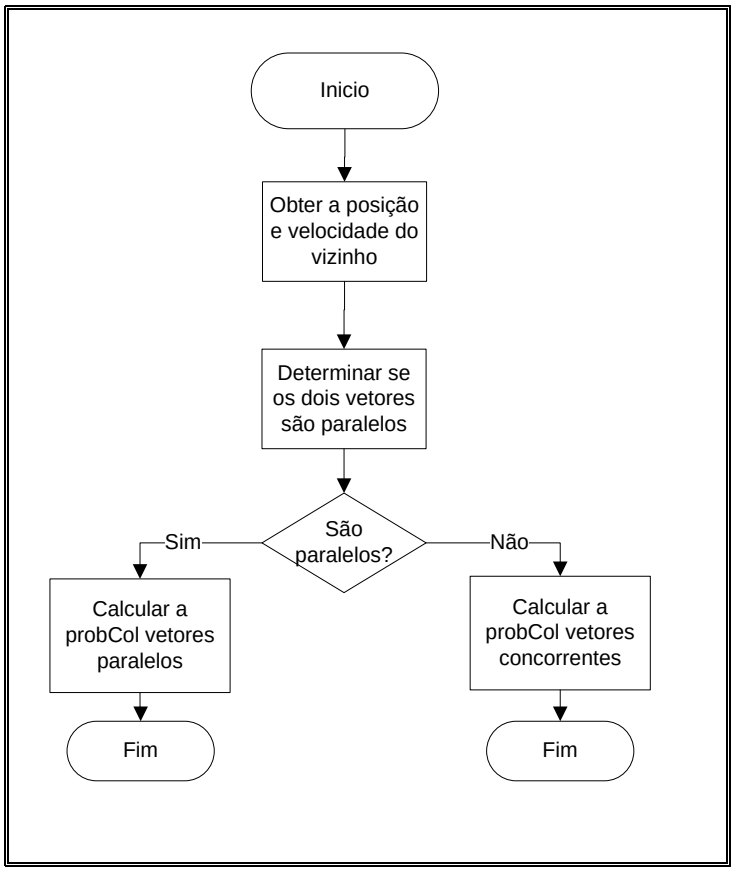


Figura 26 - Fluxograma geral do cálculo da probabilidade de colisão entre dois atores

Inicialmente, é necessário obter-se a posição e a velocidade a que se desloca o ator vizinho, estes dados encontram-se no *HashMap* que contém a lista de atores vizinhos, a *neighbourList*.

Como os atores se vão deslocar sobre linhas no mapa, através de exclusão de hipóteses, consegue-se determinar que tipo de risco o ator vizinho representa para o ator. Determinar se o vetor vizinho está a deslocar-se paralelamente e coincidente ou concorrentemente ao próprio ator, permite perceber se os dois atores estão a deslocar-se sobre a mesma linha ou em linhas diferentes. Consoante o modo como se desloca o ator vizinho em relação ao ator, pode ser num movimento paralelo ou num movimento concorrente e serão usados cálculos diferentes para determinar a probabilidade de colisão dos vetores. Na *Figura 27* temos a representação do vetor ator e o vetor ator vizinho.

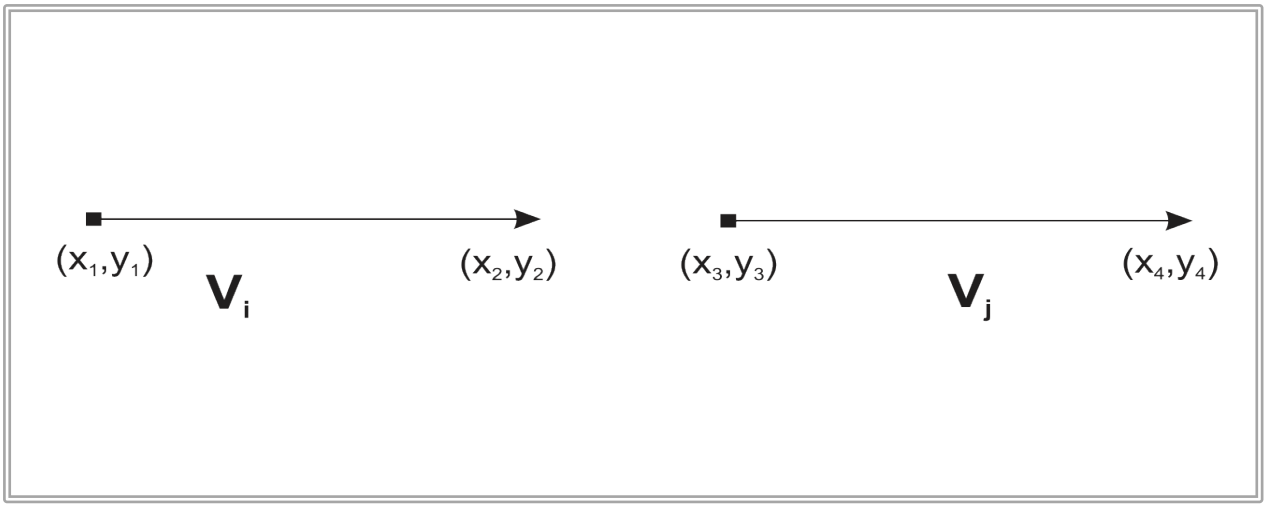


Figura 27 - Representação das coordenadas do vetor ator ($V_i$) e vetor ator vizinho ($V_j$)

Os pares de coordenadas ($x_1$,$y_1$) e ($x_3$,$y_3$) são as coordenadas atuais dos dois Actores. Os pares de coordenadas ($x_2$,$y_2$) e ($x_4$,$y_4$) são o resultado da soma das coordenadas atuais dos atores com o vetor velocidade em x e y.

$$x_2 = x_1 + v_x \qquad y_2 = y_1 + v_y \qquad \text{(Eq. 7)} \quad \text{(Eq.8)}$$

$$x_4 = x_3 + v_x \qquad y_4 = y_3 + v_y \qquad \text{(Eq.9)} \quad \text{(Eq.10)}$$

A forma de determinar se são vetores paralelos ou concorrente é pela fórmula a seguir:

$$(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1) = 0 \qquad \text{(Eq. 11)}$$

Para os dois vetores estarem em paralelo tem de se confirmar a condição anterior, ou seja, o resultado tem de ser zero. Se esta condição não se confirmar, os vetores são concorrentes.

Como foi referido antes, se forem vetores paralelos é feita uma abordagem e se forem concorrentes é usada uma outra abordagem. Serão de seguida descritas ambas as soluções, começando pelo caso de os vetores serem paralelos.

No fluxograma da *Figura 28*, está representa a solução para o caso de serem vetores paralelos.

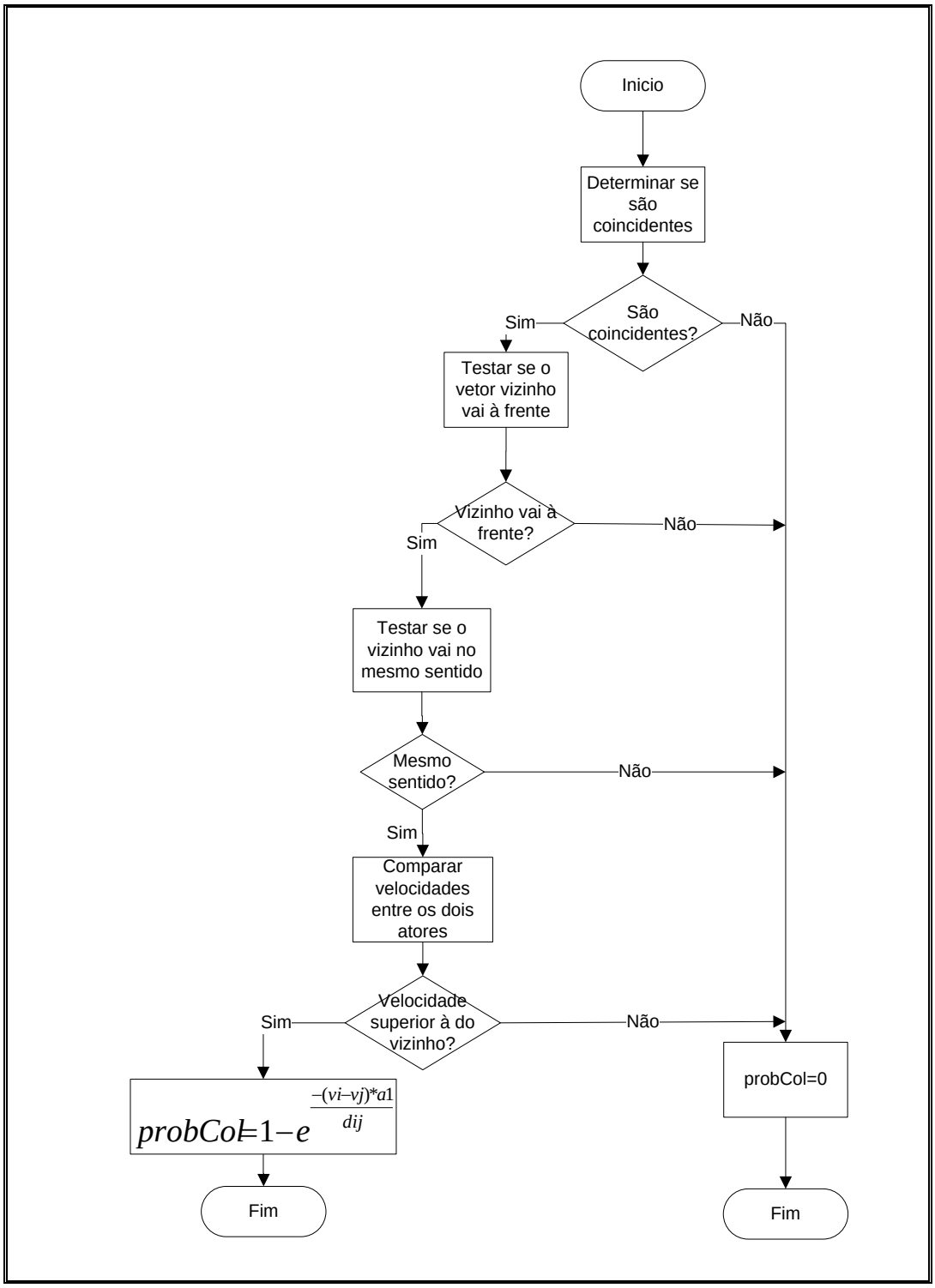


Figura 28 – Fluxograma da solução para vetores paralelos

Sendo paralelos, o primeiro passo é testar se os vetores são coincidentes, ou seja, duas retas pertencem ao mesmo plano e têm pontos em comum. Para serem coincidentes, as seguintes expressões têm de se confirmar.

$$(x_2 - x_1)(y_1 - y_3) - (y_2 - y_1)(x_1 - x_3) = 0 \quad \text{(Eq. 12)}$$

$$(x_4 - x_3)(y_1 - y_3) - (y_4 - y_3)(x_1 - x_3) = 0 \quad \text{(Eq. 13)}$$

Se forem coincidentes tem de se detetar qual dos dois vetores irá na frente. Se não forem coincidentes a probCol= 0 e não faz mais nenhum teste.

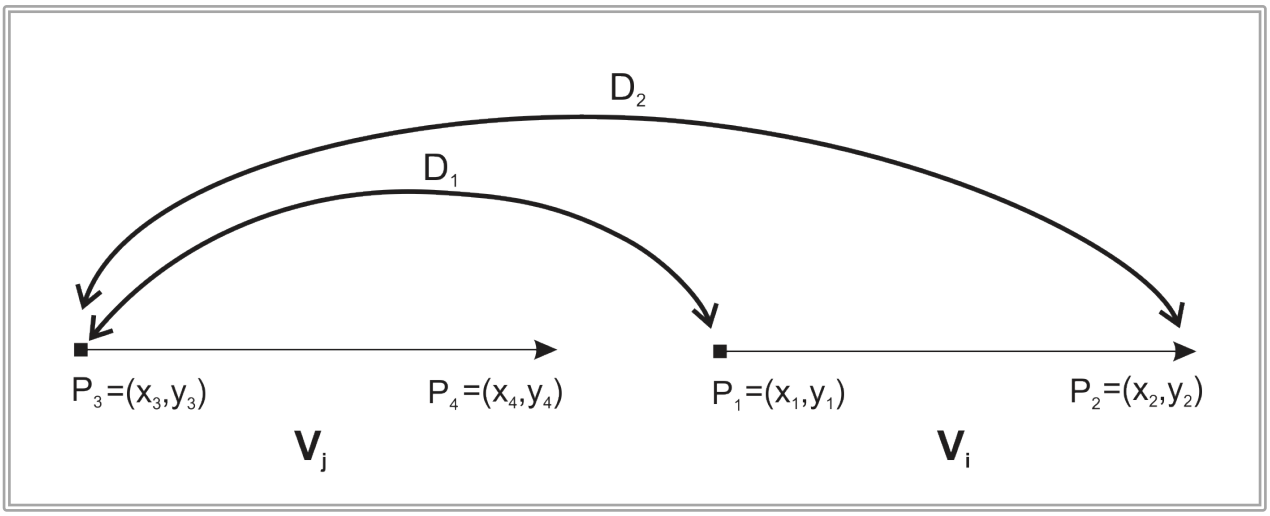


Figura 29 - Vetor vizinho (Vj) encontra-se atrás do vetor ator (Vi)

Se o ator vizinho for atrás do ator, como se pode verificar na *Figura 29*, então probCol=0, pois o ator apenas precisa de se preocupar com os atores que vão à sua frente. Para se verificar se o ator vizinho se encontra à frente do ator, como se verifica na *Figura 30,* tem de se confirmar a seguinte expressão:

$$D_2 < D_1 + D_{P_1P_2} \quad \text{(Eq. 14)}$$

em que:

$d_1$ - distância entre o inicio do vetor ator e do o inicio do vetor vizinho.

$d2$ - distância entre o fim do vetor ator e o inicio do vetor vizinho.

$d_{P1P2}$ - distância entre o inicio e o fim do vetor ator.

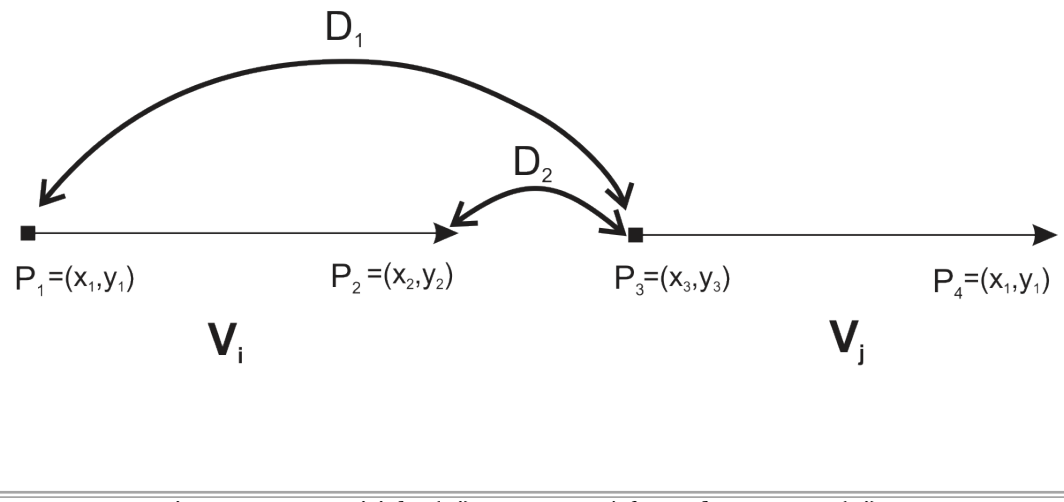


Figura 30 - Vetor vizinho (Vj) encontra-se à frente do vetor ator (Vi)

Se o ator vizinho for à frente terá de testar se os dois vetores vão no mesmo sentido.

Se os dois vetores não forem no mesmo sentido (*Figura 31*), é o caso em que $D_3>D_4$, a probCol=0. Quando $D_3<D_4$ (*Figura 32*), então os dois vetores vão no mesmo sentido.

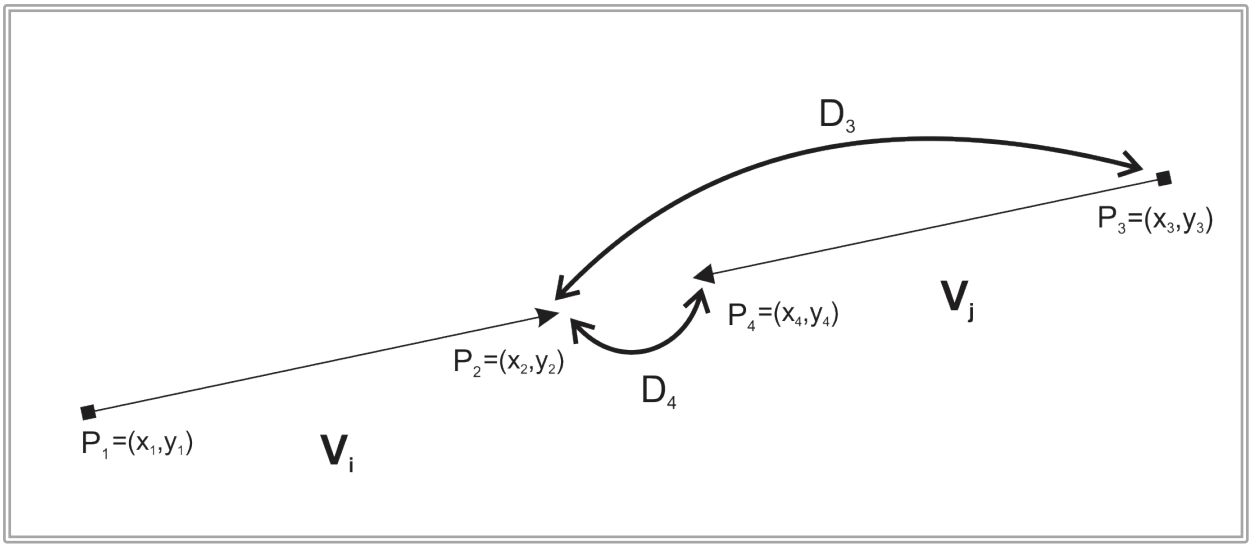


Figura 31 - Vetor vizinho (Vj) e vetor ator (Vi) vão em direções opostas

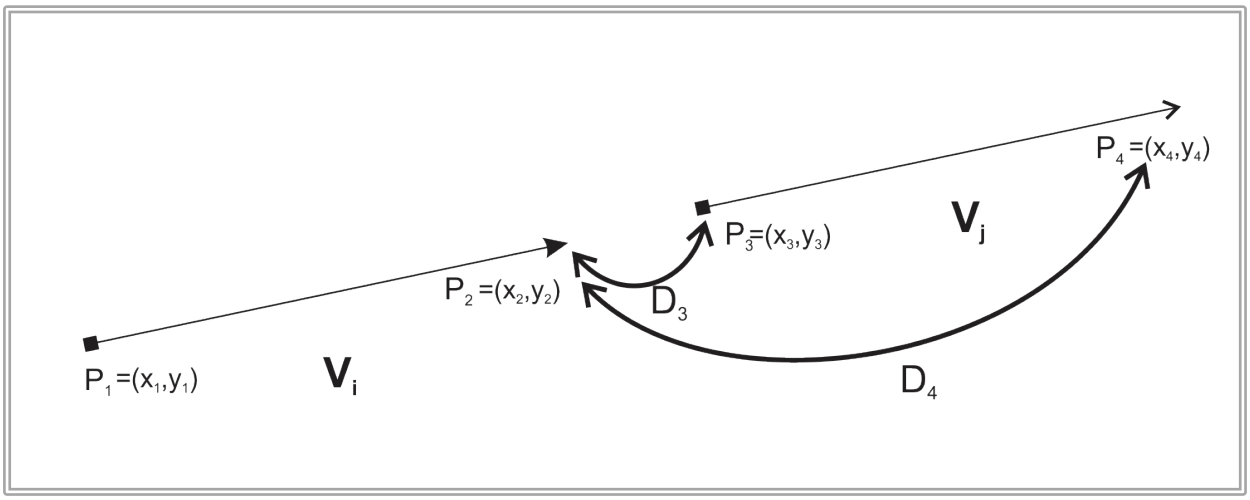


Figura 32 - Vetor ator vizinho ($V_j$) e vetor ator ($V_i$) vão no mesmo sentido

Em que:

$D_3$ - distância entre o fim do vector actor e o início do vector vizinho.
$D_4$ - distância entre o fim do vector actor e o fim do vector vizinho.

Quando vão no mesmo sentido tem de se comparar a velocidade do ator com a do ator vizinho.

Quando $V_{ator} < V_{ator\ vizinho}$ significa que nunca vão colidir porque o vetor ator não vai conseguir alcançar o vetor vizinho, por isso, probCol=0.

Quando $V_{ator} > V_{ator\ vizinho}$ existe probabilidade de colisão, sendo esta calculada pela seguinte expressão,

$$probCol = 1 - e^{\frac{-(v_i - v_j) * a1}{D_{ij}}} \quad \text{(Eq. 15)}$$

em que:

$v_i$ - velocidade do ator.

$v_j$ - velocidade do ator vizinho.

$a_1$ - constante de ajuste para o calculo.

$D_{ij}$ - distância entre o ator e o actor vizinho.

A variável probCol poderá ter o valor máximo de 1, que significa que os atores já colidiram e tem o valor mínimo de 0 que significa que não existe qualquer risco de colisão entre os atores.

De seguida, é apresentado um gráfico com o comportamento esperado para a equação 15:

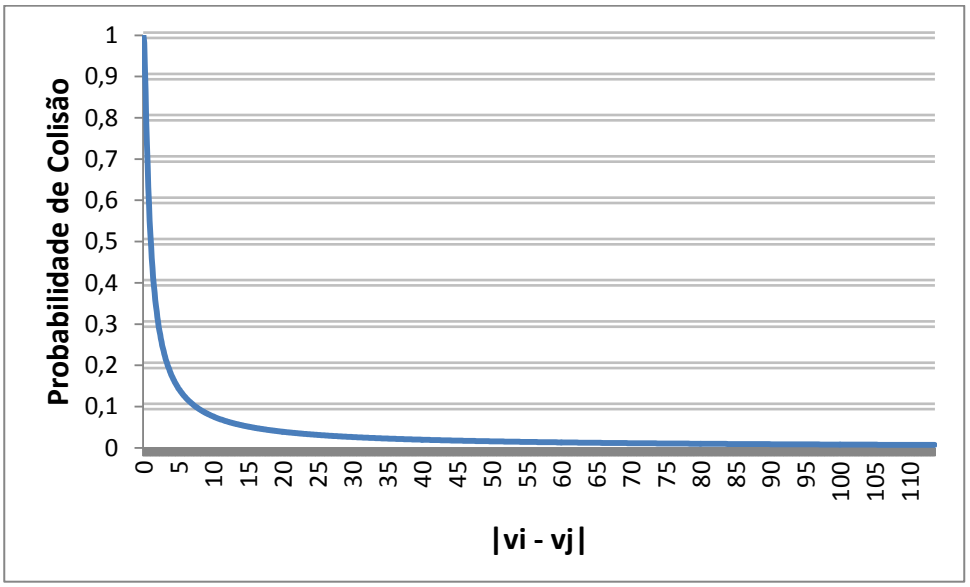


Figura 33 - Gráfico representativo da fórmula probCol da equação 15.

É apresentada, através de um fluxograma na *Figura 34*, a solução para quando os atores são concorrentes e depois uma explicação detalhada do mesmo.

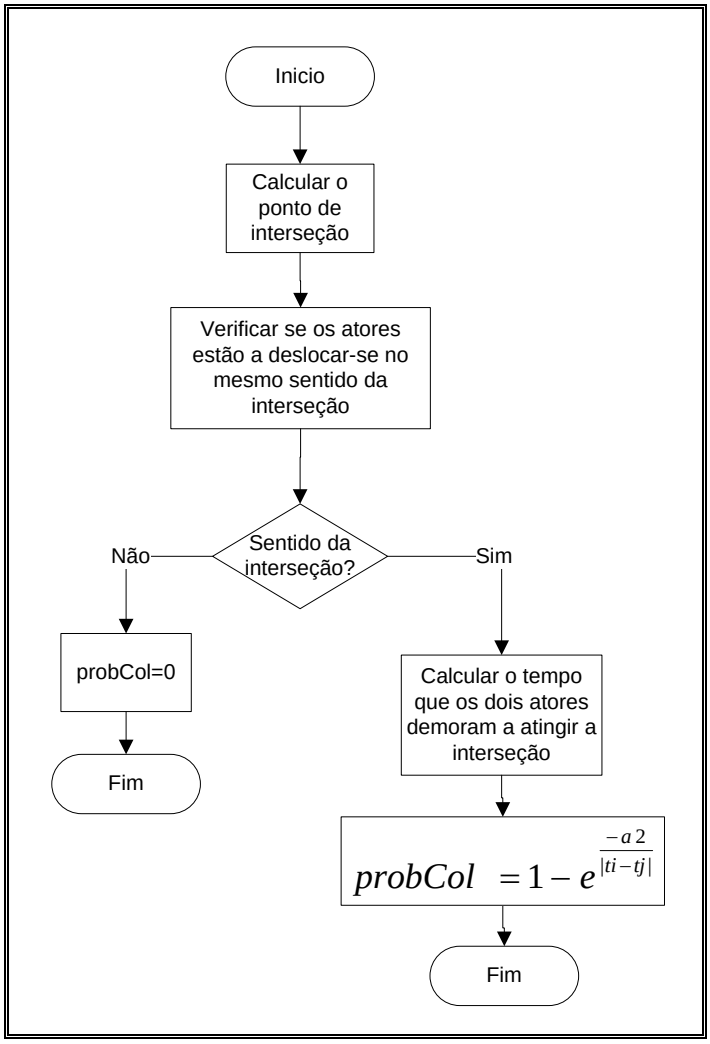


Figura 34 – Estrutura da solução para vetores concorrentes

Se os vetores forem concorrentes, significa que os dois vetores pertencem ao mesmo plano mas que existe apenas um ponto de interseção entre os dois vetores. Para determinar esse ponto de interseção temos:

$$PI_x = x_1 + u_a(x_2 - x_1) \quad \text{(Eq. 16)}$$

$$PI_y = y_1 + u_a(y_2 - y_1) \quad \text{(Eq. 17)}$$

$$u_a = \frac{(x_4 - x_3)(y_1 - y_3) - (y_4 - y_3)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)} \quad \text{(Eq. 18)}$$

em que,

$PI_x$ - coordenada do ponto de interseção em x

$PI_y$ - coordenada do ponto de interseção em y

(as outras variáveis estão representadas na *Figura 28*)

Após o cálculo do ponto de interseção (ponto PI) terá de se determinar se os dois atores se estão a deslocar no sentido do ponto de interseção (*Figura 35*). Se um dos atores não for nesse sentido (*Figura 36*), significa que não se vão encontrar em nenhum ponto, por isso probCol=0.

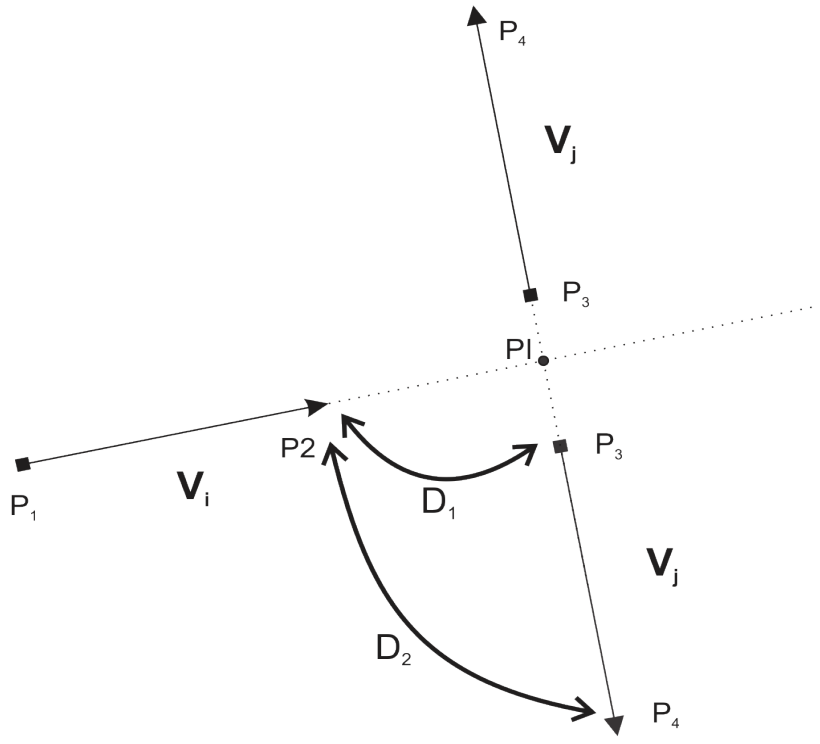


Figura 35 - Vetor ator ($V_i$) a deslocar-se no sentido do ponto de interseção e vetor vizinho ($V_j$) a ir no sentido oposto.

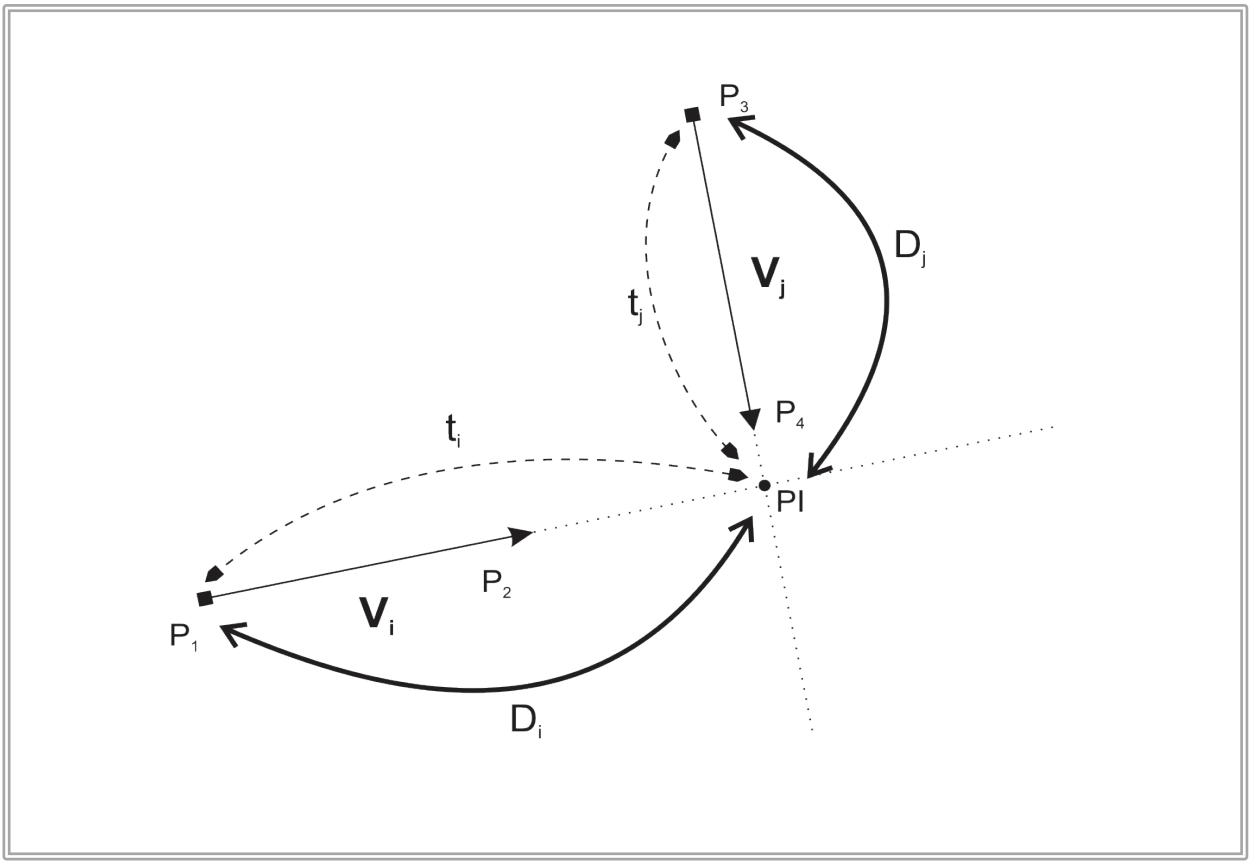


Figura 36 - Vetor ator ($V_i$) e vetor vizinho ($V_j$) a deslocarem-se no sentido do ponto de interseção

Se confirmar que os atores estão a deslocar-se no sentido da intersecção, é preciso calcular o tempo que cada ator demora a atingir o ponto de interseção. Consoante esses tempos, a probabilidade de colisão será:

$$probCol = 1 - e^{\frac{a_2}{|t_i - t_j|}} \quad \text{(Eq. 19)}$$

em que:

- $t_i$ - tempo que o ator demora a atingir o ponto de interseção.
- $t_j$ - tempo que o ator vizinho demora a atingir o ponto de interseção.
- $a_2$ - constante de ajuste para o calculo.

De seguida, é apresentado um gráfico com o comportamento esperado para a equação 19:

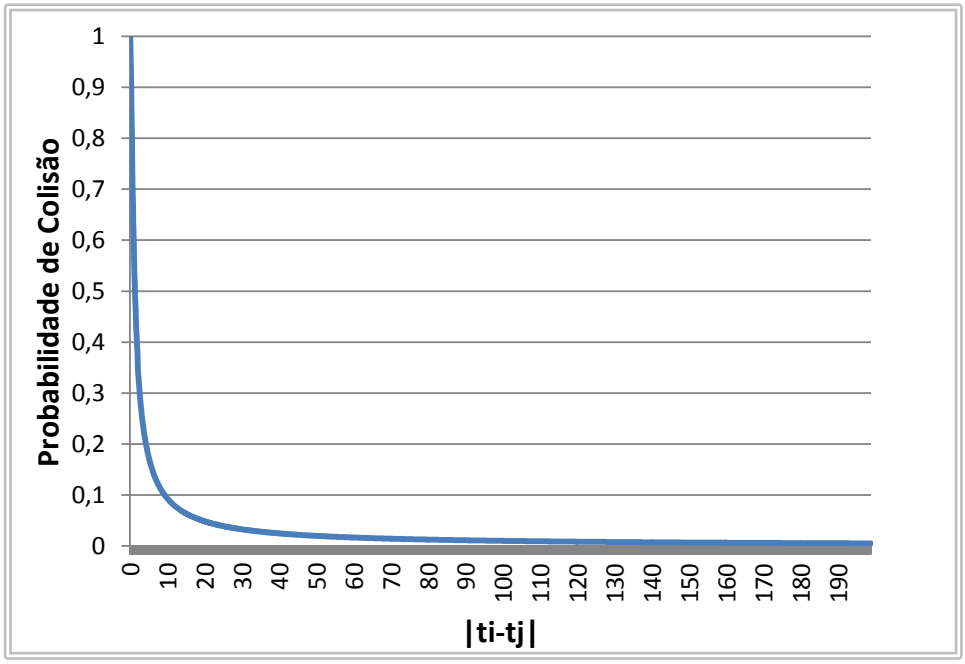


Figura 37 - Gráfico representativo da fórmula probCol da equação 19.

Uma vez calculado o valor máximo de probabilidade de colisão, tem de se atualizar o valor da velocidade do ator, para isso é usado o método *updateSpeed*.

**updateSpeed**

A atualização da velocidade (bloco 3 representado na *Figura 22)* vai depender do valor da *probCol* que se recebe. Se o valor é 0, significa que não há risco de colisão, se for um valor muito alto, superior a 0.95, tem de se atuar imediatamente e diminuir a velocidade para 0, pois os atores estão na iminência de colidir. Se o valor de *probCol* estiver entre 0 e 0.95, $0<probCol<0.95$, então a velocidade é alterada consoante o valor da *probCol*. A velocidade é calculada pela seguinte expressão:

$$velocidade = velocidade_{actual} * (1-probCol) \quad \text{(Eq. 20)}$$

em que:
velocidade - novo valor da velocidade
$velocidade_{actual}$ - valor da velocidade antes da atualização
probCol - valor da probabilidade de colisão

No caso de a *probCol* ser 0, o ator pode deslocar-se à vontade, pois não existe nenhuma possbilidade de colisão com atores vizinhos. Se a velocidade estiver a 0, o valor da velocidade é inicializado com um valor mínimo para que o ator se comece a deslocar. Se tiver um valor de velocidade superior a 0, é aumentado o valor da velocidade, com um incremento de 10% da sua velocidade atual. Para uma melhor compreensão foi elaborado o fluxograma apresentado mais à frente (*Figura 38*).

No fim de todas as atualizações de velocidade, é evocado o método *setSpeedVector* para atualizar o vetor velocidade. Este método já foi explicado anteriormente na secção 4.3.3.

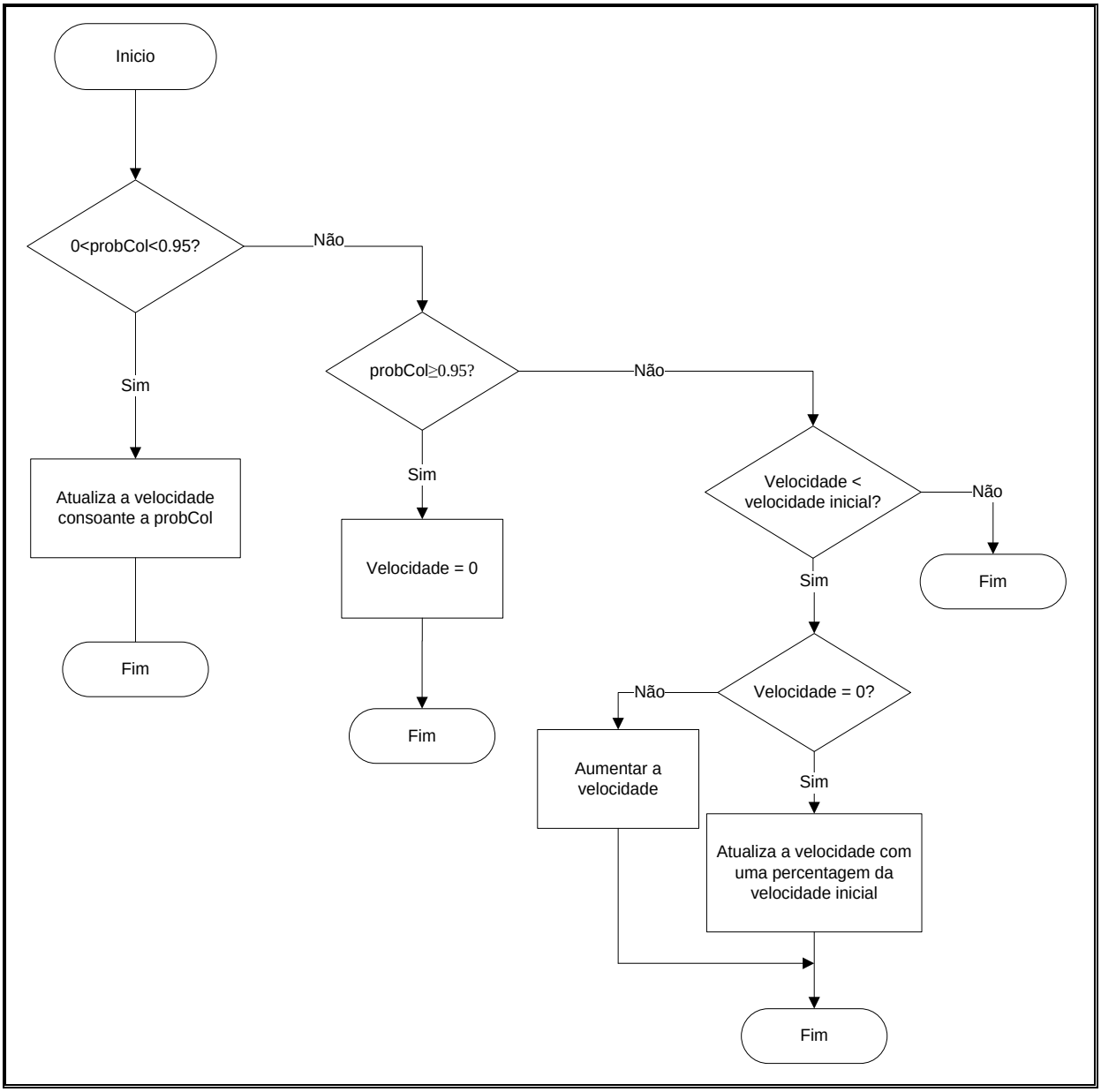


Figura 38 - Fluxograma da atualização da velocidade

## updatePosition

Uma vez atualizada a velocidade, pode-se atualizar a posição (bloco 4 representado no diagrama geral do *Actor Tram* (*Figura 22*). Com a atualização da posição do ator podem acontecer duas situações. A nova posição calculada pode-se encontrar em cima da linha do mapa que está a percorrer (caso 1 da figura seguinte), ou

pode ultrapassar o limite da linha (caso 2). No segundo caso é necessário ter o cuidado de detetar se essa situação está a acontecer e ajustar o movimento do ator para uma nova linha do mapa (caso 2: solução).

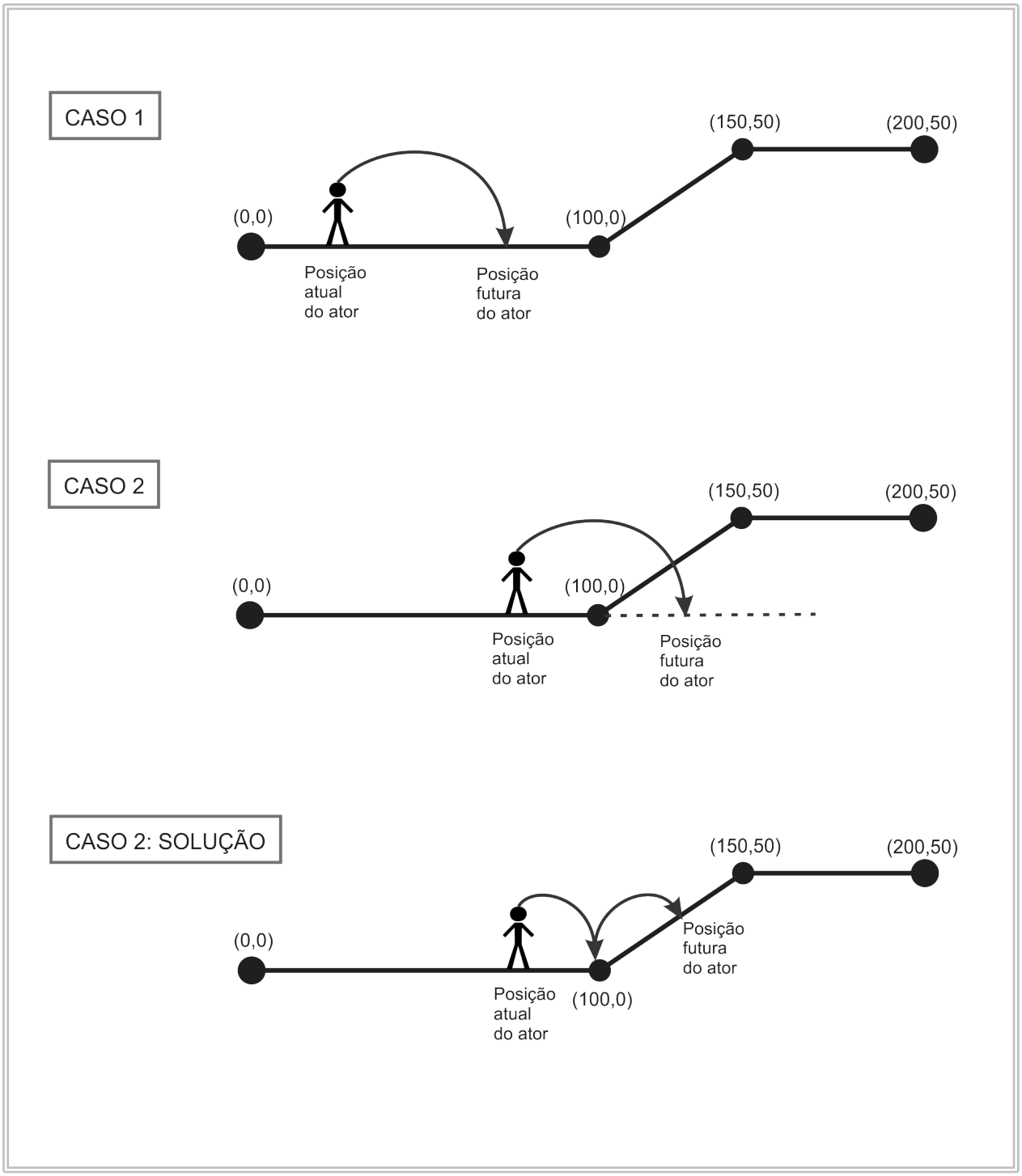


Figura 39 – Exemplo da atualização da posição do ator para uma nova linha.

A atualização da posição é feita com as seguintes expressões:

$$x_{novo} = x_{atual} + v_x * \Delta t \quad \text{(Eq. 21)}$$
$$y_{novo} = y_{atual} + v_y * \Delta t \quad \text{(Eq. 22)}$$

em que:

$x_{novo}$ - nova coordenada do ator em x

$x_{atual}$ - coordenada atual do ator em x

$y_{novo}$ - nova coordenada do ator em y

$y_{atual}$ - coordenada atual do ator em y

$v_x$ - vetor velocidade em x

$v_y$ - vetor velocidade em y

$\Delta t$ - intervalo de tempo de atualização dos atores

Para determinar se a nova posição está dentro do segmento ou se passa o limite é usada a seguinte condição:

$$(x_{novo} - x_{atual})^2 + (y_{novo} - y_{atual})^2 > (x_{fs} - x_{atual})^2 + (yfs - y_{atual})^2 \quad \text{(Eq. 23)}$$

em que:

$x_{fs}$ - coordenada final em x do segmento que o ator está a percorrer

$y_{fs}$ - coordenada final em y do segmento que o ator está a percorrer

Se a condição anterior não se verificar, significa que o ator continua sobre o mesmo segmento, por isso atualiza-se a posição normalmente. Se a condição se confirmar é necessário adaptar o ator para o novo segmento, para evitar que saia de cima do persurso no mapa. Para isso, o ator é colocado no fim do segmento atual, as coordenadas x e y ficam com as coordenadas do fim de segmento. De seguida, é necessário saber qual o novo segmento que o ator irá percorrer e, como se sabe qual o

início do segmento (coordenada atual do ator), é possível determinar o próximo destino do ator através do método *findNextDestination()* (explicado anteriormente) para determinar qual o novo destino do ator. Uma vez sabendo qual o novo segmento de reta que vai percorrer, é necessário recorrer ao método *setSpeedVector()* para ajustar a direção e sentido do vetor velocidade de acordo com o segmento que vai percorrer. Como o movimento do ator ultrapassava o limite do segmento anterior, é necessário calcular esse excesso e acrescentá-lo no novo segmento, para evitar que o movimento do ator não seja completo.

Após atualização da posição acaba o método moveActor, é atualizado o *SimStatus,* faz uma pausa e depois recomeça o ciclo.

### *4.4.4 Generator Tram*

O *Generator Tram* será semelhante ao *Generator Random*, também cria atores com base nos valores que se encontram no ficheiro de configuração mas, neste caso, como os atores irão precisar de mapas para se movimentar, será necessário incorporar na mensagem de criação de atores, que é enviada do *GlobalCoordinator* para o *LocalCoordinator*, os nomes dos mapas que o ator irá precisar para se movimentar, bem como os mapas com os pontos de paragens do *Tram,* sendo que no *Actor Tram* atuais são ignoradas as paragens (ainda não foi implementada essa parte).

A cadência a que os Actors Tram são gerados será muito baixa, e será necessário restringir o número destes atores que poderão estar em funcionamento durante a simulação para dar um maior realismo à mesma, como acontece na realidade em que os valores de *trams* a circular numa cidade são sempre reduzidos.

## 4.5 Ator Carro

Um carro é um veículo que se desloca apenas sobre estradas. Ao contrário do tram*,* o carro pode ultrapassar outros carros se a estrada tiver mais do que uma faixa de rodagem. Pode estacionar por pouco tempo ou por muito tempo, ou pode, simplesmente

parar indefinidamente. As estradas em que o carro se desloca têm regras que têm de ser observadas, como limites de velocidade e sentidos de trânsito. Existem diversos tipos de condutores que têm diferentes comportamentos. Existem condutores que andam sempre dentro dos limites de velocidade, e têm uma condução mais ao menos segura, outros condutores que têm comportamentos mais agressivos, como andar acima do limite de velocidade ou fazerem ultrapassagens quase em cima dos outros carros. Normalmente, os carros são as entidades que se deslocam mais depressa no ambiente urbano.

### *4.5.1 Requisitos*

O *Actor Car* também deverá ter um comportamento semelhante ao modelo *City Section Mobility Model* e *Manhattan Mobility Model*. O ator terá de se deslocar sobre mapas pré definidos e sempre que chegar a um cruzamento deverá fazer a escolha de qual caminho seguir. O ator tem de ter em conta todos os outros atores que o rodeiam, tomando decisões tais como ultrapassar, acelerar e abrandar, tem de atualizar o seu estado periodicamente e reportar o seu estado para o *SimStatus*.

### *4.5.2 Modelo Comportamental*

De acordo com as especificações apresentadas, foram idealizados alguns comportamentos para o *Actor Car*.

O *Car* tem de andar sobre as linhas do mapa, como o *Actor Tram*, mas com a diferença que poderá sair da linha temporariamente quando está a ultrapassar uma outra entidade. O carro, sendo a entidade que se desloca mais depressa, é importante definir que a sua velocidade normalmente será superior ao resto das entidades da simulação. Se o ator não conseguir ultrapassar outro, tem de abrandar para não colidir com ele, assim como acontece no *Actor Tram*.

Como existem diferentes tipos de condutores, é importante definir diferentes comportamentos para o *Actor Car*, o comportamento normal, agressivo e, como o *Car*

pode estacionar, então é definido um comportamento diferente para quando o ator quiser estacionar.

Este ator tem de cumprir algumas regras de trânsito, como limite de velocidade, limite de faixas de rodagem ou estradas com um ou dois sentidos.

Tal como na vida real, os carros podem parar por muito tempo ou sair da cidade, por isso é importante definir um tempo limite para um ator permanecer na simulação, mas tendo esse tempo de ser diferente para todos os atores.

### *4.5.3 Implementação*

Depois de pensados os requisitos e o modelo comportamental, foi estruturada uma abordagem, de modo a conseguir respeitar essas caraterísticas.

Quando o *Actor Car* é criado, são carregados todos os parâmetros, como acontece nos atores anteriores. Uma das diferenças mais relevantes do *Actor Car* para os atores anteriores, é o facto do *Actor Car* ter tempo de vida. O tempo de vida é usado para simular as situações na vida real, como quando algum carro sai da cidade ou um carro estaciona por muito tempo. Nestas duas situações, não se justifica estar a gastar processamento com atores que não vão voltar à simulação, assim sendo, o ator "morre" e é eliminado da simulação. O tempo de vida é decrementado a cada atualização do ator.

Após as configurações iniciais o ator é colocado no mapa com o método *findNextDestination()* e é criado o vetor velocidade para o ator com o método *setSpeedVector()*, ambos os métodos explicados na secção *4.4.3*. O novo ator é guardado no *SimStatus* e é evocado o método *moveActor* para começar o movimento.

De seguida, será explicado com mais detalhe o método *moveActor* do *Actor Car*. No fluxograma a seguir, pode-se ver a estrutura geral da abordagem usada para implementar o comportamento do *Actor Car*.

#### ***moveActor***

No fluxograma da *Figura 40*, os blocos foram numerados de forma a ser mais fácil a identificação de cada bloco. A estrutura do método *moveActor* do *Actor Car* é um

pouco diferente da estrutura do *Actor Tram*. Este ator tem apenas um bloco novo, o bloco 1.

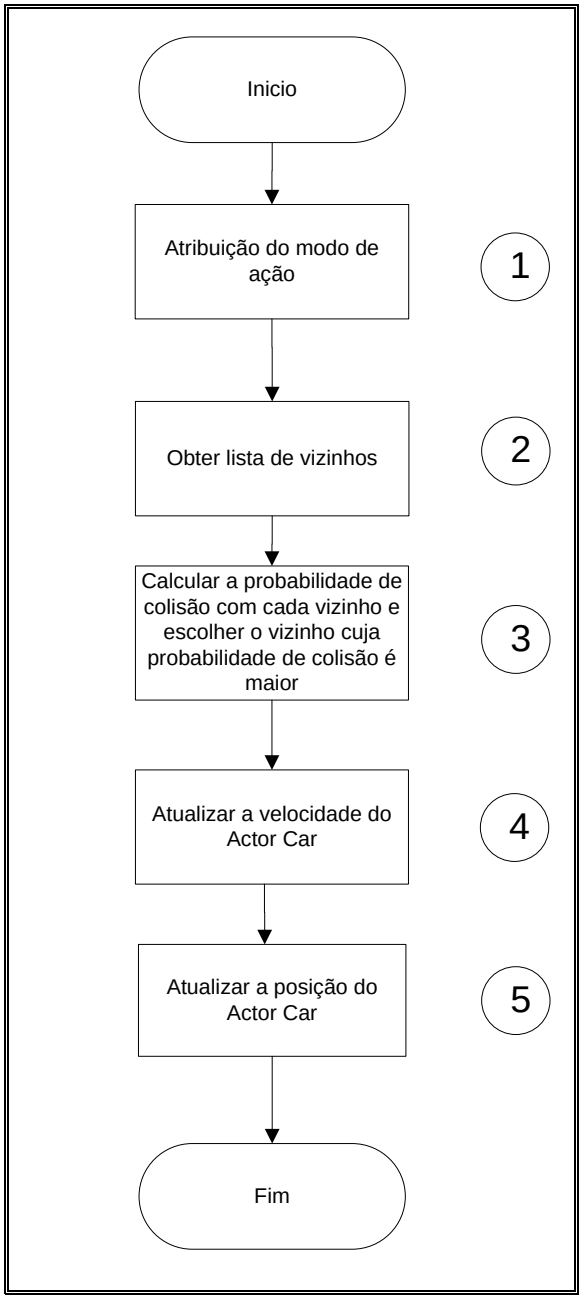


Figura 40 – Fluxograma geral do método moveActor do Actor Car

No bloco 1 é atribuído um modo de ação para o ator e, dependendo do modo atribuído, o *Actor* terá um comportamento diferente.

Os outros blocos têm a mesma função que os blocos apresentados no *Actor Tram*, mas com um comportamento diferente. No caso do bloco 2 e 3, são exatamente iguais aos blocos 1 e 2 do fluxograma geral do *Actor Tram*.

Com o fluxograma representado na *Figura 41,* pretende-se especificar mais o fluxograma apresentado anteriormente para um melhor entendimento da estrutura do método *moveActor*. No fluxograma é referido *mode = 2*, isto apenas será explicado mais a frente, mas significa que o ator vai estacionar ou já estacionou.

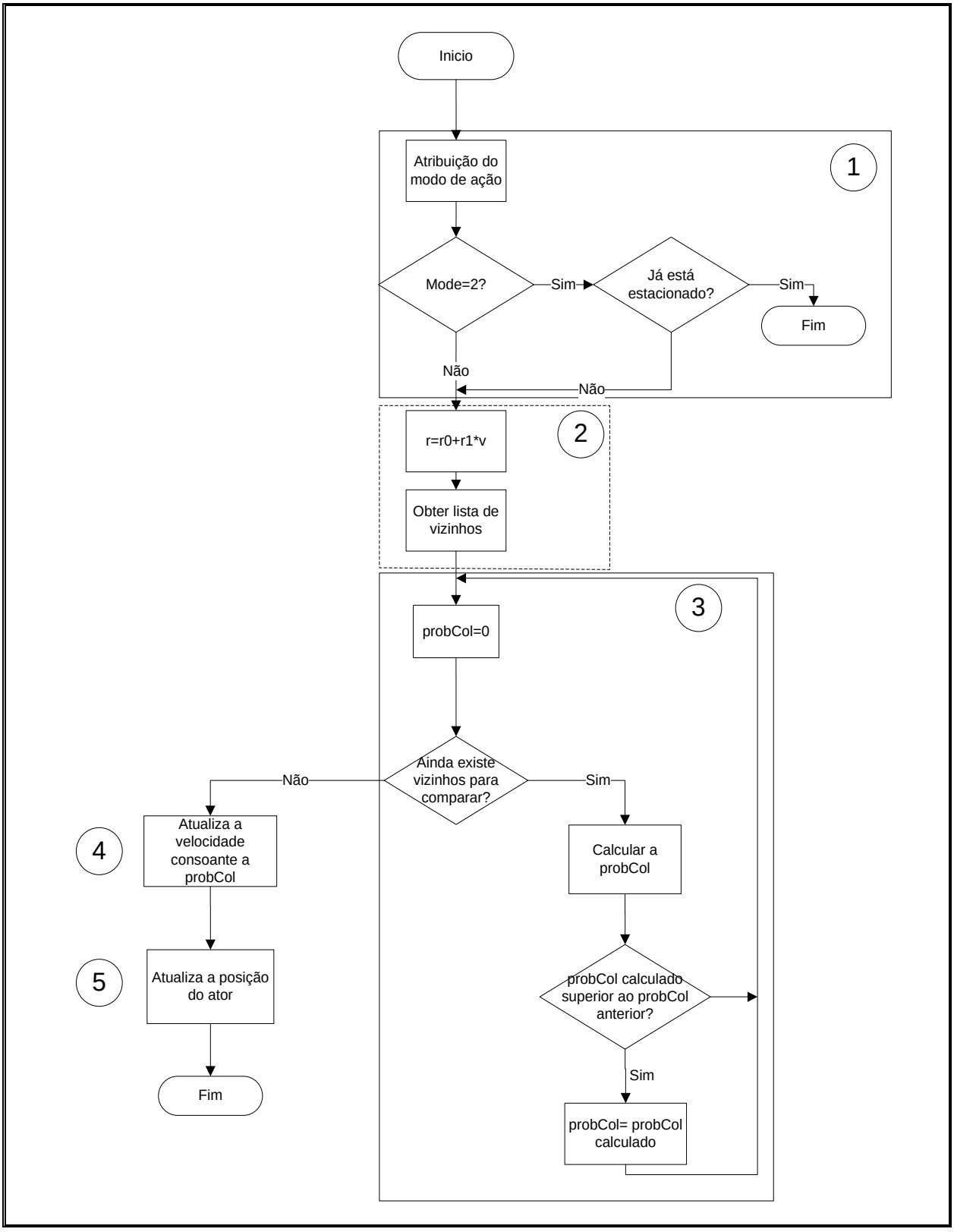


Figura 41 - Fluxograma geral do *Actor Car*

**setMode**

De seguida, é definido o modo de ação do ator (bloco 1 da *Figura 40*), no método *setMode*. Os modos existentes vão alterar o comportamento de condução do *Actor Car*. Foram implementados diferentes modos de comportamento, nomeadamente *modo normal (mode=0)*, o *modo agressivo (mode=1)*, o *modo estacionamento (mode=2)* e o modo a estacionar (*mode*=3).

No *modo normal,* o comportamento do *Actor* tem de respeitar as regras de condução impostas, como por exemplo, o limite de velocidade. No *modo agressivo,* o comportamento do condutor pode não respeitar as regras de condução impostas, por exemplo, não vai respeitar o limite de velocidade ou fará travagens mais bruscas. No *modo estacionamento*, é quando o ator decide estacionar em algum lugar. Quando o ator decide estacionar existe um momento em que o ator inicia o processo de estacionamento até estacionar, aí será definido como *modo a estacionar*. Para ajudar na compreensão da mudança de modo, foi criado o seguinte diagrama de estados (*Figura 42*).

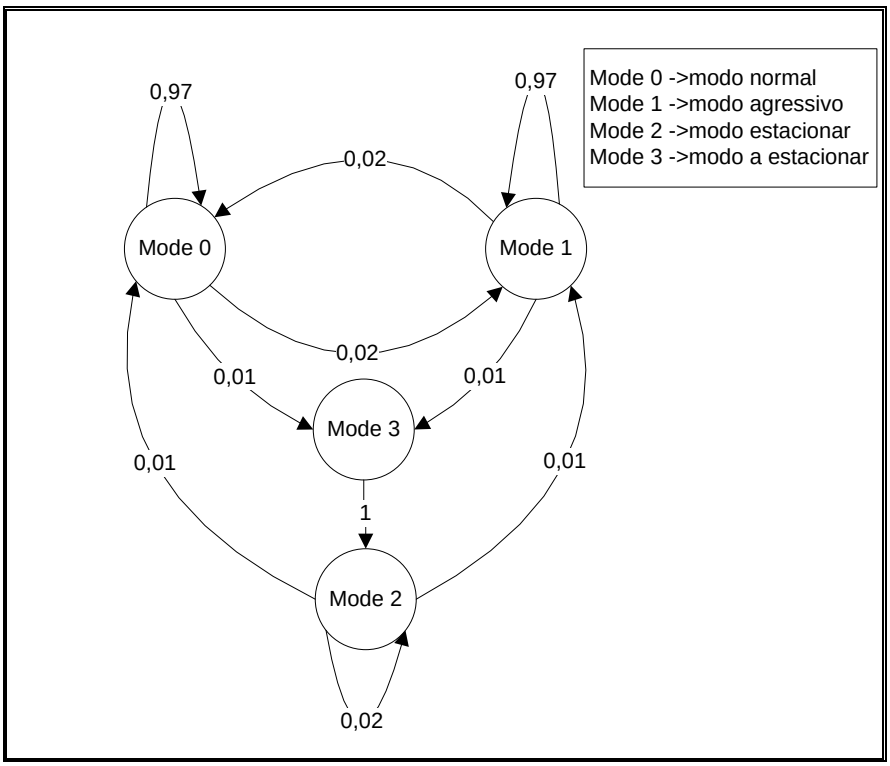


Figura 42 - Diagrama de estado do método *setMode*.

Inicialmente o ator começa no modo normal, depois, consoante uma dada probabilidade, pode mudar de modo. Para que o ator não esteja constantemente a mudar de comportamento, é dada uma probabilidade baixa para mudar de modo. É gerado um número aleatório entre 0 e 1 e são definidos intervalos para mudança de modo. Se o valor estiver entre [0,0.97], o estado atual do ator não sofre alterações. Se o valor estiver entre ]0.97,0.99], o ator muda o estado de *normal* para *agressivo* ou dos modos *agressivo* ou *a estacionar* para o estado *normal*. Se o valor estiver entre ]0.99,1], então, o ator pode mudar para o modo a *estacionar* ou *agressivo*. Ou seja, se estiver no modo *normal* ou *agressivo* o ator passa para modo *a estacionar*. Se não estiver em nenhum dos dois significa que está estacionado e vai começar a mover-se, passando para o modo *normal*. Todas as alterações dependem do estado em que o ator se encontra no momento. Para um melhor entendimento, no fluxograma a seguir, é apresentada a estratégia adoptada para a atribuição dos diferentes modos:

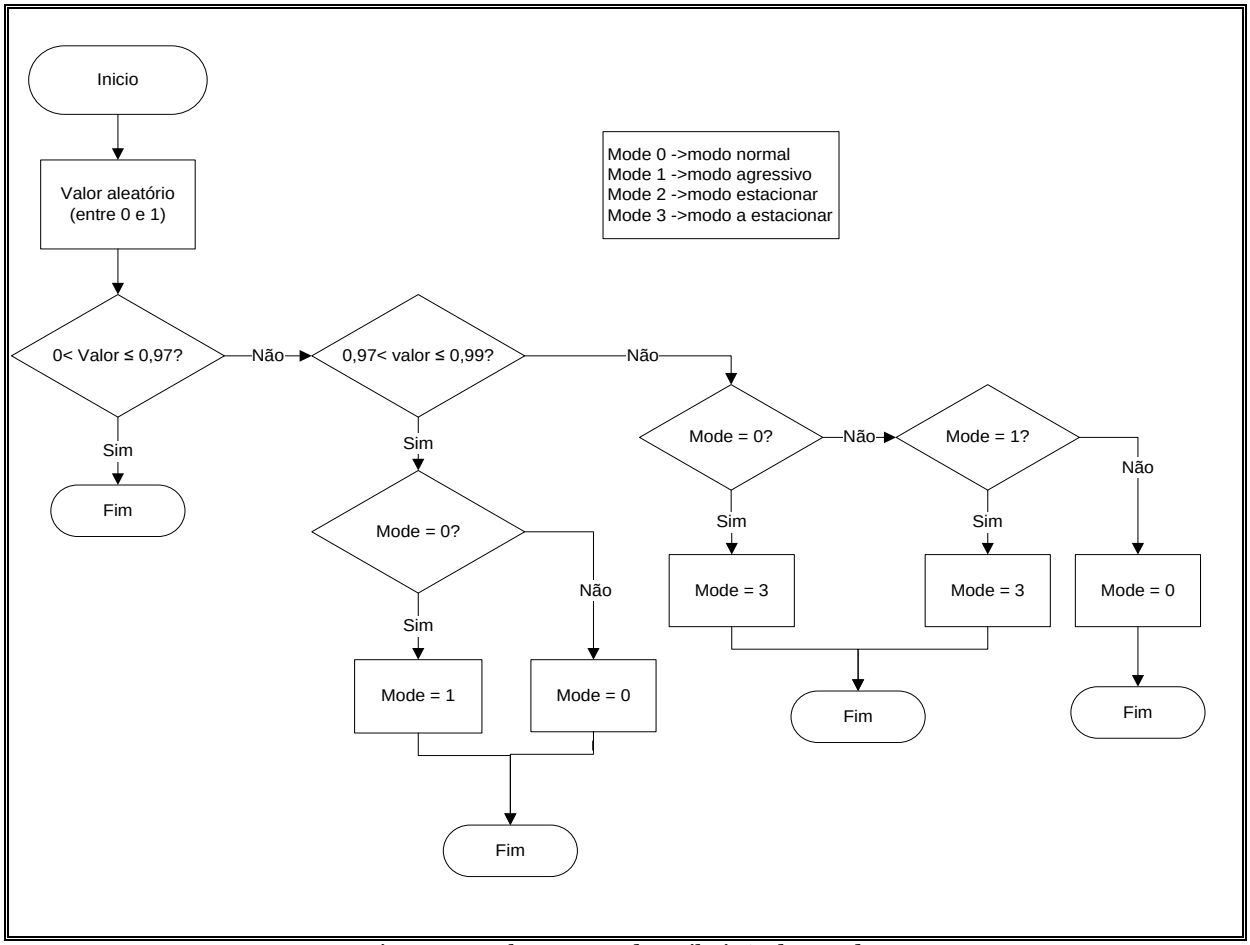

Figura 43 – Fluxograma de atribuição dos modos

Após ter sido atribuído o modo de comportamento ao ator, se este estiver no modo de estacionamento, não faz mais nada até que o seu modo volte a mudar. Esta decisão foi tomada de maneira a poupar recursos de processamento, visto que o ator está parado e não se justifica estar a gastar recurso com funções que não vão alterar o estado do ator. Caso contrário, é obtida a lista de vizinhos e é calculada a probabilidade de colisão com cada um deles. Como já foi dito, é feito da mesma maneira que no *Actor Tram*. Após o cálculo da *probCol,* é ajustada a velocidade consoante essa mesma probabilidade e de seguida é atualizada a posição do ator de acordo com a velocidade. Estes dois últimos passos também existem no *Actor Tram*, mas foram definidos diferentes comportamentos. De seguida será explicado com mais detalhe cada um deles.

**updateSpeed**

De modo a facilitar a explicação do método *updateSpeed,* foi criado um fluxograma com a estrutura da implementação e, de seguida, é explicado o mesmo (*Figura 44*):

Como o *Actor Car* pode ultrapassar e estacionar, é neste ponto que é decidido se pode ultrapassar ou se não pode. Inicialmente é testado se o ator está parado. Se estiver, não é preciso testar se vai ultrapassar e não faz a atualização da velocidade consoante a *probCol*. Consoante o modo em que se encontra o ator tem diferentes abordagens à ultrapassagem. A necessidade de ultrapassar depende do valor da *probCol*, no caso do *modo normal*, a *probCol* mínima para ultrapassa será de 0.7, ou seja, quando estiver muito perto do ator vizinho tenta ultrapassar. No *modo agressiv*o, ele vai tentar ultrapassar muito mais cedo, o valor mínimo de *probCol* é de 0.2.

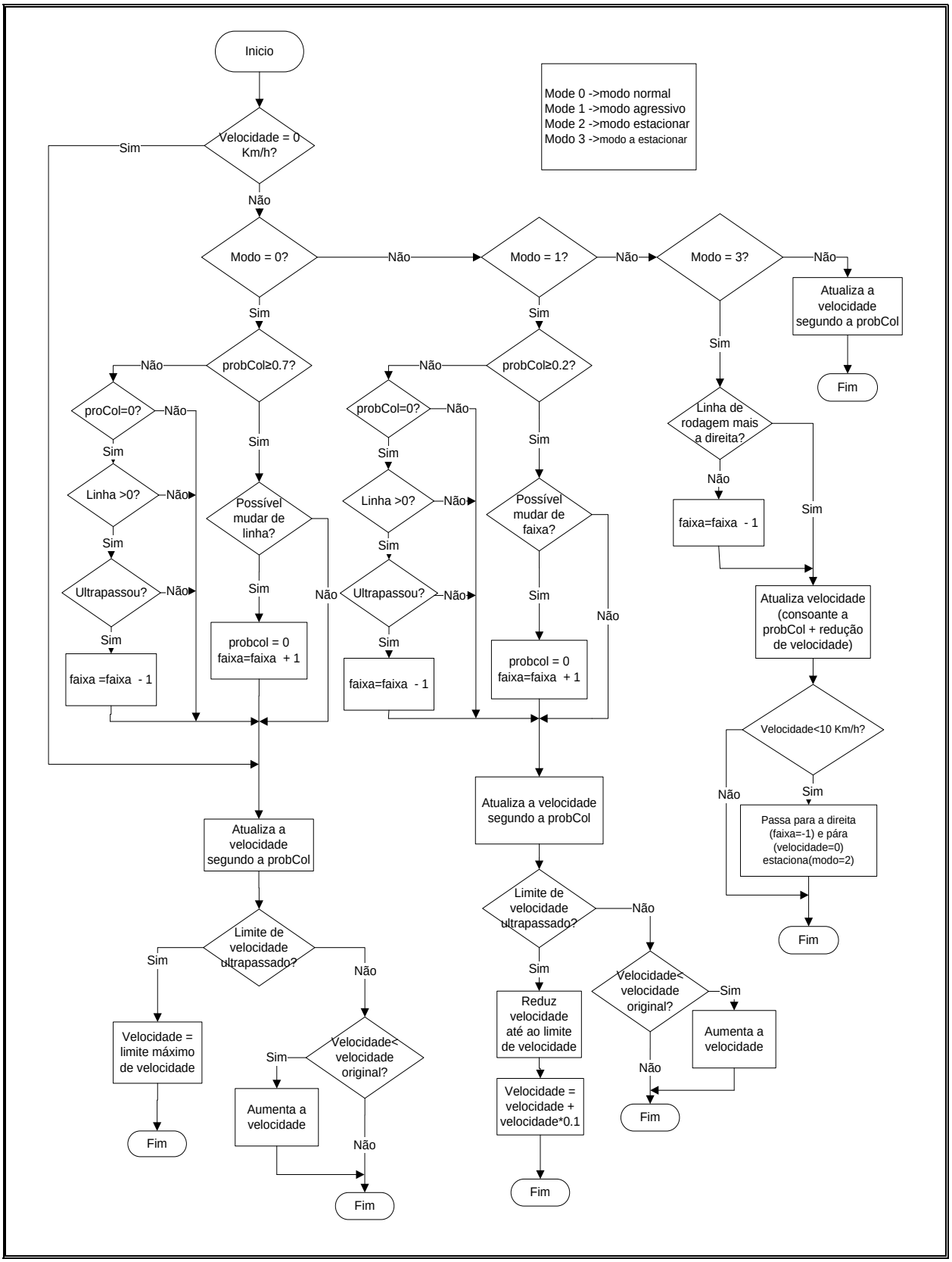


Figura 44 - Fluxograma geral do Actor Tram

Se o valor da *probCol* for superior aos limites anteriores nos dois modos, o ator vai tentar ultrapassar. Para o ator poder ultrapassar é necessário que a via tenha mais do que uma faixa de rodagem do que aquela em que o ator se encontra naquele momento. Inicialmente o ator começa a deslocar-se na faixa 0. Por exemplo, a via pode ter três faixas, o ator pode ultrapassar se estiver nas duas primeiras faixas; se estiver na terceira faixa não pode ultrapassar. Se o ator ultrapassar, então coloca-se a *probCol* a 0, incrementa-se a faixa em que o ator se encontra. Se não puder ultrapassar, mantém-se na mesma faixa e continua com a mesma *probCol*. Para se determinar quanto tempo o ator fica a ultrapassar, numa primeira versão, será usado um contador que é inicializado quando inicia a ultrapassagem e, quando atingir um valor pré definido tenta voltar para a faixa de rodagem de origem. O contador é incrementado a cada atualização do ator.

Se o valor da *proCol* não passar os valores mínimos para ultrapassar, significa que não precisa de ultrapassar e que pode ser necessário passar para a faixa de rodagem mais à direita. Para isso acontecer é preciso que a *probCol* esteja a 0 e, que o ator não se encontre na faixa de rodagem mais à direita (faixa> 0). Se estas condições forem cumpridas o ator desloca-se uma faixa de rodagem para a direita.

Depois de definida a faixa de rodagem em que o ator vai-se movimentar, é atualizada a velocidade dependendo da *probCol*, como já foi descrito no *Actor Tram*. De seguida, é testado se a velocidade do ator está a ultrapassar a velocidade limite da via. Senão ultrapassar a velocidade limite, é testado se a velocidade do ator é inferior à sua velocidade original, quando foi criado. Se for inferior, a velocidade é aumentada, da mesma maneira que o *Actor Tram,* no modo *normal*. No modo *agressivo* o aumento de velocidade é mais rápido; se não for inferior à sua velocidade inicial, mantém-se a velocidade.

Quando está no *modo normal,* se a velocidade limite da via for ultrapassada, a velocidade do ator diminui para o limite máximo. Se estiver no *modo agressivo* a velocidade do ator é aumentada até ser superior em 10% ao limite de velocidade.

Quando o ator está em *modo estacionamento,* o objetivo é fazer abrandar o ator, passar para a faixa de rodagem mais à direita (faixa=0) e só depois parar na faixa -1 que representa uma linha fora da zona de deslocação dos atores. Inicialmente, testa-se para saber se o ator se encontra na faixa de rodagem mais à direita, se não estiver, decrementa a faixa de rodagem, de seguida atualiza a velocidade de acordo com a

*probCol,* mais uma redução extra para aos poucos ir diminuído a velocidade. Depois testa se a velocidade é baixa o suficiente para estacionar o carro, se sim, pára o carro e muda a faixa de rodagem para -1. Uma vez defininda a sua velocidade e faixa de rodagem é atualizada a posição do ator no método *updatePosition()*.

**updatePosition**

A atualização da posição do ator é muito semelhante à atualização do *Actor Tram*, com a diferença que pode ter de atualizar fora das linhas pré definidas do mapa quando está a fazer uma ultrapassagem.

O primeiro teste que tem de fazer, assim como no Actor Tram, é calcular se a próxima posição vai ultrapassar o limite do segmento em que o ator se encontra. Se ultrapassar, tem de colocar o ator no limite do segmento, procurar um novo destino e atualizar de seguida o vetor velocidade do ator. Se não passar o limite, apenas atualiza a posição do ator.

Para atualizar a posição são usadas as expressões apresentadas a seguir, como acontece no *Actor T*ram:

$$X_{temp} = x_{atual} + v_x * \Delta t \quad \text{(Eq. 24)}$$

$$Y_{temp} = y_{atual} + v_y * \Delta t \quad \text{(Eq. 25)}$$

em que:

$x_{temp}$ - próxima posição do ator em x

$x_{atual}$ - coordenada atual do ator em x

$y_{temp}$ - próxima posição do ator em y

$y_{atual}$ - coordenada atual do ator em y

$v_x$ - vetor velocidade em x

$v_y$ - vetor velocidade em y

$\Delta t$ - intervalo de tempo de atualização dos atores

Como no momento da atualização o ator pode estar numa outra faixa de rodagem, é necessário colocá-lo na faixa de rodagem original. Após ser calculada a nova posição, o

ator será colocado numa linha paralela ao segmento em que o ator se encontra naquele momento. A distância a que o ator vai ficar do segmento, depende da faixa de rodagem em que o ator se encontra.

Foi criado o fluxograma a seguir, para apoio da explicação da mudança de faixa de rodagem do ator (*Figura 45*):

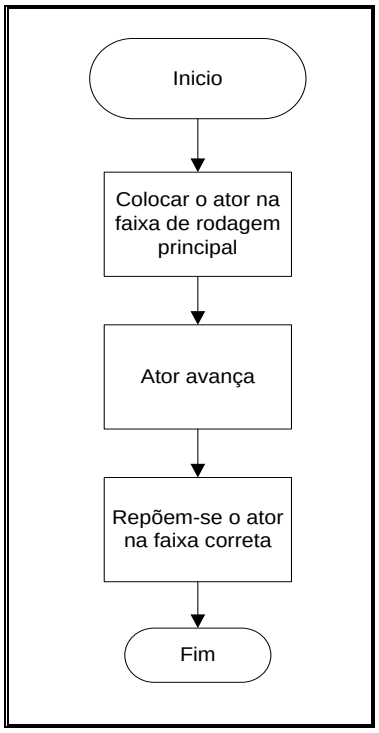


Figura 45 - Fluxograma do modo de atualização da posição do Actor Car.

Para colocar o ator na faixa correta, foram usadas as seguintes fórmulas:

$$X_{novo} = X_{temp} - \frac{faixa * V_y}{\sqrt{V_x{}^2 + V_y{}^2}} \quad \text{(Eq. 26)}$$

$$Y_{novo} = Y_{temp} + \frac{faixa * V_x}{\sqrt{V_x{}^2 + V_y{}^2}} \quad \text{(Eq. 27)}$$

em que:

$x_{temp}$ -coordenada temporária do ator em x

$x_{novo}$ -nova coordenada do ator em x

$x_{atual}$ -coordenada atual do ator em x

$y_{temp}$ -coordenada temporária do ator em x

$y_{novo}$ -nova coordenada do ator em y

$y_{atual}$ -coordenada atual do ator em y

$v_x$ -vetor velocidade em x

$v_y$ -vetor velocidade em y

faixa -faixa de rodagem que o ator se encontra

Após a atualização da posição do *Actor Car*, acaba o método *moveActor*, é atualizado o *SimStatus* com os novos valores, faz uma pausa e depois recomeça o ciclo.

### *4.5.4 Generator Car*

O *Generator Car* é muito semelhante ao *Generator Tram*, pois também cria atores com base nos valores que se encontram no ficheiro de configuração. O intervalo de velocidades a que cada ator irá entrar na simulação terá valores maiores do que no *Actor Tram*.

A cadência que os *Actors Cars* são gerados será mais alta e o número de atores que poderão estar ativos durante a simulação será muito mais elevado que no caso do

*Actor Tram*, conseguindo assim dar um maior realismo à simulação, pois, o número de carros a circular numa cidade é bastante elevado.

## 4.6 Ator Pedestre

Andar, sempre foi o principal meio de locomoção dos humanos e, é também o meio mais complexo e difícil de modelar.

Um pedestre, normalmente, pode deslocar-se por todo lado numa cidade, no passeio, na estrada e também passar próximo de carris. A velocidade de um pedestre é a mais baixa de todas as entidades, numa cidade. Por norma o pedestre anda sempre na mesma direção que o passeio, mas o movimento do ator não é sempre paralelo ao passeio onde se desloca, tem movimentos mais irregulares. Não são movimentos totalmente aleatórios, nem totalmente lineares. A velocidade do ator não é sempre constante, existem picos de velocidade quando o pedestre está com pressa, uma diminuição quando o pedestre esta a passear, e também paragens curtas se o ator estiver a ver uma montra ou, longas, se estiver à espera de um autocarro.

Os pedestres compõem uma grande parte das entidades numa dada cidade. Por norma o número dos pedestres é muito superior ao número de carros a circular.

### *4.6.1 Requisitos*

O *Actor Pedestrian* também deverá ter um comportamento semelhante ao modelo *City Section Mobility Model* e *Manhattan Mobility Model*. O ator terá de se deslocar sobre mapas pré definidos, e sempre que chegar a um cruzamento, fazer uma escolha por qual caminho deve seguir. O ator pedestre não terá de obedecer completamente às linhas pré definidas dos mapas, pois, o pedestre pode andar em qualquer lado, mas, de forma a ter um movimento realista, o seu movimento terá sempre como referência o mapa existente. Tem de atualizar o seu estado periodicamente e reportar o seu estado para o *SimStatus*.

### *4.6.2 Modelo Comportamental*

A pensar nas especificações anteriores, foi idealizado um comportamento para o Actor Pedestrian. Devido ao elevado número de pedestres que irão existir na simulação, é importante definir um comportamento simples, apesar do elevado grau de dificuldade em emular o movimento de um pedestre. Comportamentos simples conseguem criar um padrão relevante e aproximado da realidade.

O Actor Pedestrian terá como referência, para o seu movimento, um mapa igual ao dos atores anteriores, mas o movimento não estará restrito apenas às linhas do mapa, terá uma área para fazer a sua movimentação. A área será criada com uma linha paralela à linha do mapa, a uma certa distância. As duas linhas representarão os limites laterais ao movimento do ator, assim como acontece na realidade nos passeios destinados aos pedestres.

A velocidade a que Actor Pedestrian se vai deslocar será mais ao menos constante. A cada atualização do ator, a velocidade sofrerá uma ligeira alteração, ou seja, a velocidade está sempre em alteração mas essa variação não é muito percetível. Essas alterações apesar de impercetíveis, podem levar um dado ator a atingir uma velocidade caraterística de uma pessoa que está com pressa e vai com uma velocidade acima da média ou, pode levar o ator a parar ou andar muito devagar. Com isso, o Actor Pedestrian poderá ter diversos comportamentos dependendo de uma certa aleatoriedade.

### *4.6.3 Implementação*

Depois de pensados os requisitos e o modelo comportamental, foi estruturada uma abordagem de modo a conseguir respeitar essas caraterísticas.

Quando o *Actor Pedestrian* é criado, são carregados todos os parâmetros, como acontece nos atores anteriores. Assim como acontece no *Actor Car*, o *Actor Pedestrian* também irá ter tempo de vida. Cada ator terá um tempo de vida diferente, calculado aleatoriamente dentro de um intervalo de tempo igual ao do *Actor Car*.

O ator é colocado no mapa com o método *findNextDestination* e é criado o vetor velocidade para o *Acto*r com o método *setSpeedVector*, ambos os métodos explicados na secção 4.5.3. O novo *Actor* é guardado no *SimStatus* e é evocado o método *moveActor* para começar o movimento.

De seguida, será explicado com mais detalhe o método *moveActor* do *Actor Pedestrian*. No fluxograma a seguir pode-se ver a estrutura geral do comportamento implementado do *ActorPedestrian (Figura 46)*.

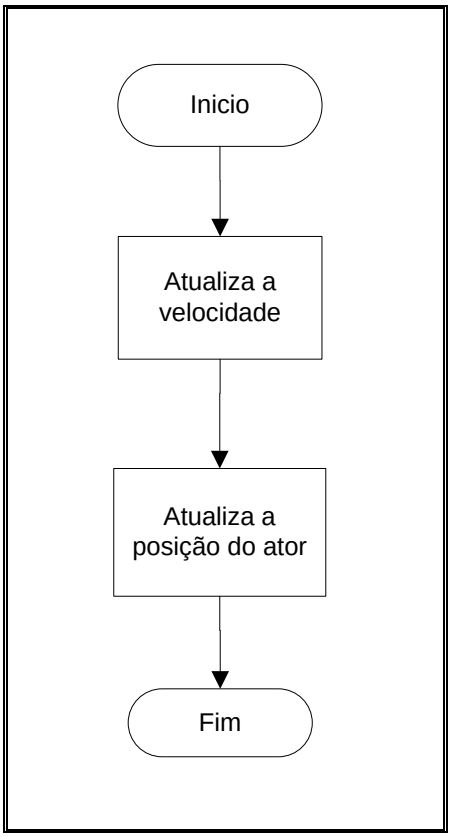


Figura 46 - Fluxograma do funcionamento do Actor Pedestrian

Neste ator não haverá diferentes modos de comportamento, nem os atores vão calcular a probabilidade de colisão com atores vizinhos. A velocidade do *Actor Pedestre* vai estar sempre a variar. A variação de velocidade vai ser pequena, mas irá criar uma certa aleatoriedade no comportamento do ator. A atualização da velocidade é feita usando a seguinte fórmula:

$$vel_{nova} = \text{vel}_{atual} + \ variação_{mínima} + r * \ (variação_{máxima} - \ variação_{mínima})$$

(Eq. 28)

em que:

$\text{vel}_{\text{nova}}$ - novo valor de velocidade do ator

$\text{vel}_{\text{atual}}$ - velocidade em que se encontra o ator antes de atualizar

$\text{Variação}_{\text{mínima}}$ - valor mínimo do intervalo que a velocidade do ator pode variar

$\text{Variação}_{\text{máxima}}$ - valor máximo do intervalo que a velocidade do ator pode variar

r - variável aleatória no intervalo entre 0 e 1

(de referir que as variações mínimas e máximas podem ser valores negativos ou positivos)

Após atualizada a velocidade, é usado o método *setSpeedVector* para atualizar o vetor velocidade do ator. De seguida, é atualizada a posição do ator usando o método *updatePosition*.

**updatePosition**

Este método é muito semelhante ao que é usado no *Actor Car* mas, com a diferença de que, a variação do ator não será apenas nas faixas de rodagem existentes. O ator pode deslocar-se em linhas paralelas ao segmento em que se encontra. Será criado um limite mínimo e um limite máximo onde o ator se poderá movimentar. Com esta estratégia, tenta-se simular o passeio onde os pedestres se deslocam normalmente.

A atualização da posição do *Actor Pedestrian* é semelhante à atualização do *Actor Car,* com a diferença que é gerado um valor aleatório para uma linha paralela em que o ator se vai deslocar. Para evitar que o ator faça movimentos muito bruscos durante a deslocação, a variação entre as linhas tem de ser pequena. Para o cálculo do novo ponto, para onde o ator se vai colocar, é usada a seguinte expressão:

$$linha_{nova} = \text{linha}_{atual} + variação_{mínima} + r * (variação_{máxima} - variação_{mínima})$$

(Eq. 29)

em que:

$\text{Linha}_{\text{nova}}$ - nova linha que o ator se vai deslocar

$\text{Linha}_{\text{atual}}$ - linha que se encontra o ator antes de atualizar

$\text{Variação}_{\text{mínima}}$ - valor mínimo do intervalo que o ator pode variar

$\text{Variação}_{\text{máxima}}$ - valor máximo do intervalo que o ator pode variar

r - variável aleatória no intervalo entre 0 e 1

(de referir que as variações mínimas e máximas podem ser valores negativos ou positivos)

Após atualização da posição do *Actor Pedestrian*, acaba o método *moveActor*, é atualizado o *ActorStatus* com os novos valores, faz uma pausa e depois recomeça o ciclo.

### *4.6.4 Generator Pedestrian*

O Generator Pedestrian é muito semelhante ao *Generator Tram e Car*, pois também cria atores com base nos valores que se encontram no ficheiro de configuração. O intervalo de velocidades a que cada ator irá entrar na simulação terá valores muito menores do que no *Actor Tram* e *Car*, pois, o pedestre é a entidade que existe em maior quantidade na simulação.

A cadência que os *Actors Pedestrian* são gerados, será mais alta, e o número de atores que poderão estar em funcionamento durante a simulação será mais elevado que o caso do *Actor Car,* conseguindo assim dar um maior realismo à simulação, pois o número de pedestres a circular numa cidade é bastante elevado.

# 5 Testes e Análise de resultados

Um dos objetivos principais do simulador BartUM é realizar simulações em grande escala. Para se conseguir atingir simulações com milhares de atores é necessário que cada ator não exija um grande esforço computacional sobre a máquina que o está a executar. Neste capítulo serão apresentados os testes efetuados e os resultados obtidos em relação ao desempenho dos atores durante uma simulação. Foram efetuados testes aos vários atores e registados os recursos computacionais necessários para os executar. Foram também feitos testes de carga a cada tipo de ator, e um teste final, com os atores *Tram*, *Car* e *Pedestian* a interagirem entre eles. No final dos resultados é feita uma análise comparativa entre os vários atores.

## 5.1 Ambiente de teste

Para realizar os testes foram usados 5 computadores com caraterísticas diferentes. As máquinas possuem algumas diferenças de modo a testar o simulador em máquinas com diferentes capacidades computacionais e com diferentes versões do sistema operativo Windows. Na tabela a seguir são enumeradas as caraterísticas mais relevantes dos computadores.

| Computador ID | Processador | Ram | Disco | IP | Sistema Operativo |
|---|---|---|---|---|---|
| Computador 1 | Pentium III 996MHz | 256MB | 7,85 GB | 192.168.0.1 | Windows XP Professional |
| Computador 2 | Intel Core I3 3.07GHz 3.06 GHZ (Quad core) | 4 GB | 465 GB | 192.168.0.2 | Windows 7 Professional (32bits) |

| | | | | | |
|---|---|---|---|---|---|
| Computador 3 | Intel Core I3 3.07GHz 3.06 GHZ (Quad core) | 4 GB | 465 GB | 192.168.0.3 | Windows 7 Professional (32bits) |
| Computador 4 | Pentium III 551MHz | 512 MB (256MB + 256MB) | 7,85 GB | 192.168.0.4 | Windows XP Professional |
| Computador 5 | Pentium III 551MHz | 256 MB | 7,85 GB | 192.168.0.5 | Windows XP Professional |

Figura 47 - Caraterísticas mais relevantes dos computadores envolvidos na simulação

Foram instalados nos computadores os sistemas operativos de novo, de forma a não existir gastos extra de outros programas e processos desnecessários existentes no computador. Criaram-se ficheiros *.jar* para cada uma das entidades da simulação de forma a poderem ser executados pela linha de comandos, diminuindo assim a carga necessária para executar o simulador. De forma a ser possível executar os ficheiros *.jar* foi instalado o *Java Runtime Environment (JRE)* versão 7.0 em cada um dos computadores. Os computadores foram configurados de modo a usarem o mínimo de recursos dos sistemas operativos, diminuindo assim o erro nos resultados obtidos durantes os testes.

Foram usadas 5 máquinas ligadas entre si por um *Switch Ethernet a 10 Mbps*, de forma a criar uma rede local, destinada apenas à troca de mensagens entre as entidades da simulação.

Com todas as máquinas do sistema, em que haverá um *GlobalCoordinator*, quatro *LocalCoordinators* e um *Visualization*, com a seguinte distribuição de funções como representado na *Figura 48*.

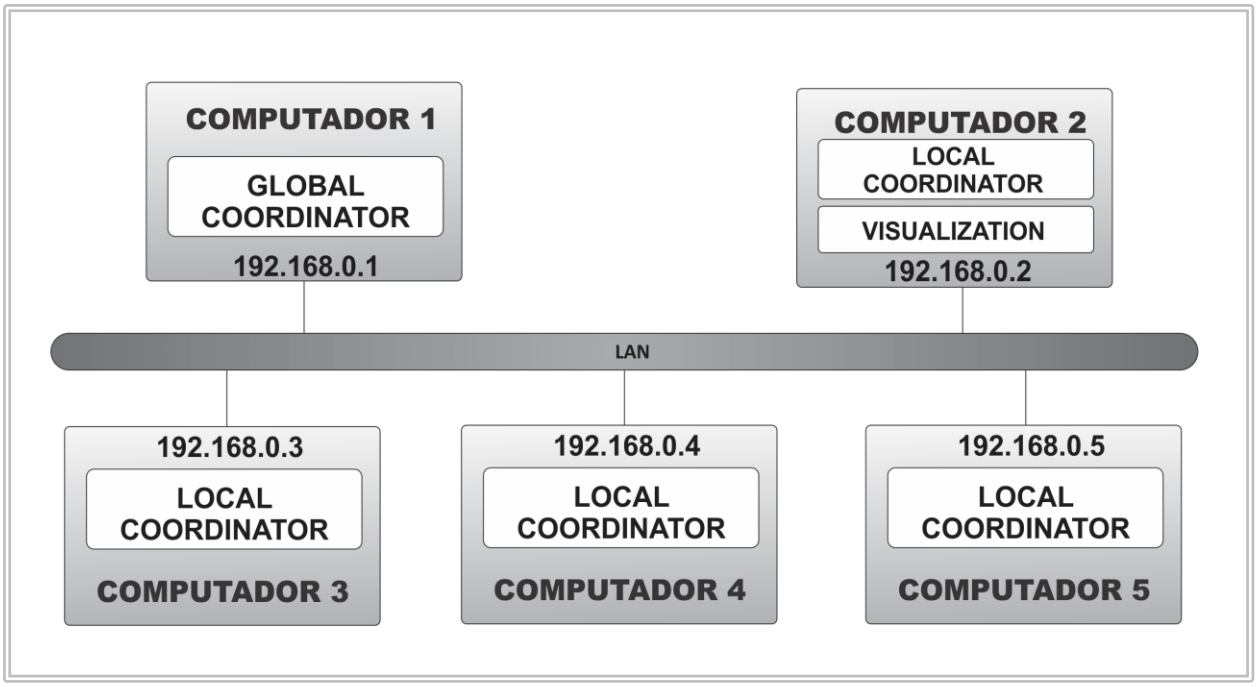


Figura 48 - Cenário de teste.

Como os computadores 2 e 3 têm caraterísticas iguais, foi atribuído ao computador 2 a *Visualization*, esperando que se obtenha valores de carga computacional superiores ao computador 3.

Foi usado um mapa OSM de uma zona parcial da cidade de Braga. O mapa foi descarregado do servidor do OpenStreeMaps, o que permitiu escolher a área pretendida para a simulação. Usando o editor de mapas do OSM, foram eliminados pontos e linhas que não fossem necessários para a simulação, corrigindo assim alguns defeitos que existiam no mapa, como por exemplo linhas que ficaram por acabar ou que ficaram a meio devido a seleção da área pretendida e ficaram incompletas.

Foram feitas as devidas configurações no ficheiro de *properties*, como o endereço *IP* da máquina que será o *GlobalCoordinator*, de modo a que todos os *LocalCoordinators* e a *Visualization* se possam ligar. Identificou-se qual o mapa que seria usado, enumeraram-se os geradores que seriam usados e o seu tipo. Por fim foram descritas as caraterísticas de cada gerador, o local onde iriam gerar os seus atores e a cadência a que os atores seriam gerados. Para a simulação foram utilizados 9 geradores, que apenas geravam atores do mesmo tipo nos testes individuais de cada ator. No último teste aos geradores foram atribuidos diferentes tipos de atores o *Tram*, o *Car* e o

*Pedestrian*, como se pode ver no extrato do ficheiro de *properties* usado na simulação apresentado a seguir.

Foi usado o mesmo cenário de simulação para todos os testes efectuados. Os testes foram feitos a cada ator individualmente e num último teste foram usados todos os diferentes atores na mesma simulação.

Na *Figura 49* pode-se ver o mapa que foi usado. Foram assinalados com pontos os locais onde os geradores estariam a criar atores.

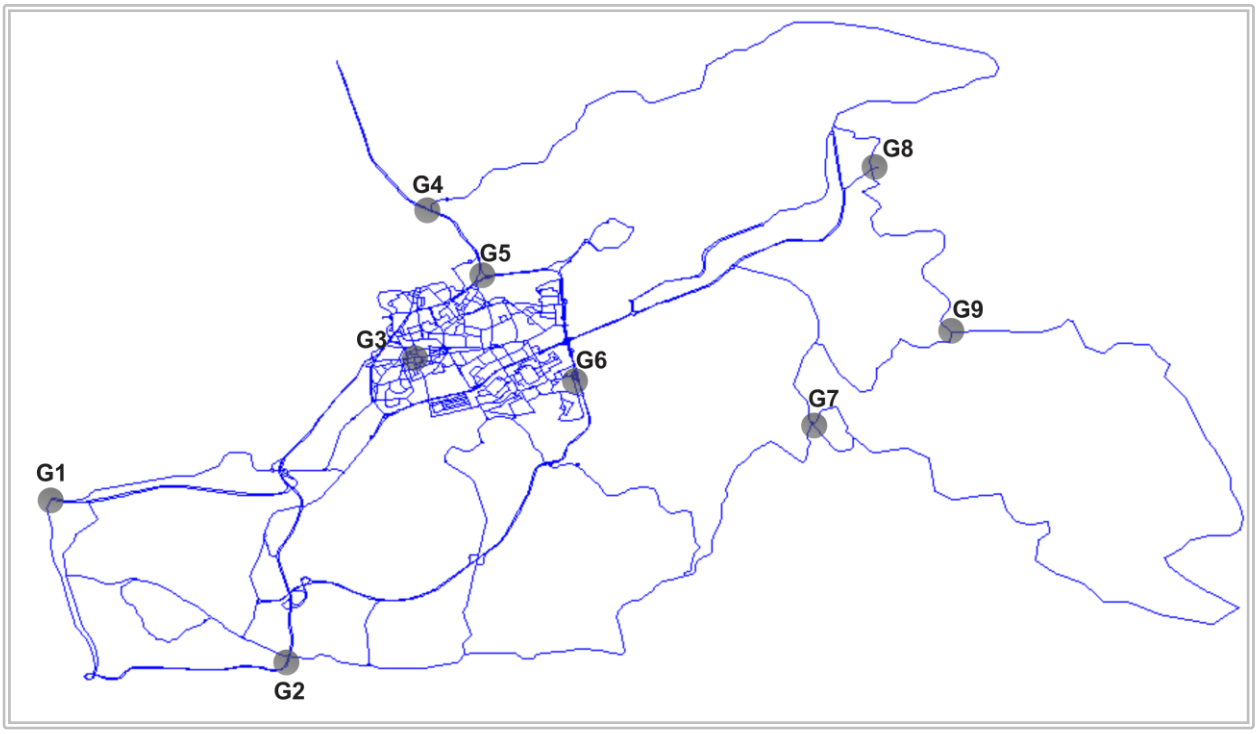


Figura 49 - Mapa usado na simulação

De forma a tornar possível uma análise correta dos resultados obtidos é necessário saber o estado inicial de cada máquina. Na tabela da *Figura 50* foram registados os dados inicias dos computadores usados.

| | Computador 1 | Computador 2 | Computador 3 | Computador 4 | Computador 5 |
|---|---|---|---|---|---|
| Processador | 0% | 0% | 0% | 1% | 1% |
| Memória RAM (ocupada) | 43% | 21% | 22% | 45% | 18% |

Figura 50 - Valores inicias do uso do processador e RAM dos vários computadores

## 5.2 Cenários de teste

De seguida, serão apresentados todos os resultados dos testes feitos a cada um dos diferentes atores. Cada resultado é divido em duas partes, uma com o desempenho do procesador e a outra os gasto de memória RAM de cada computador. Ao fim de cada cenário de teste é feita uma análise a cada resultado obtido.

### *5.2.1 Cenário de teste 1: Actors Random*

No caso do *Actor Random*, foram criados dois cenários de testes. Num primeiro teste foram usadas todas as máquinas de forma a testar o desempenho dos atores nos computadores mais fracos. No segundo teste foram usadas as 3 melhores máquinas de forma a extender o número de atores na simulação e testar o desempenho dos computadores com uma carga superior.

Como já foi explicado, o *Actor Random* terá movimentos totalmente aleatórios, não irá precisar de mapas para se deslocar. Como não existe necessidade de colocar os atores no mapa, todos os *Generator Random* vão criar os *Actors Random* nas mesmas coordenadas.

O *Actor Random*, sendo o ator mais simples, espera-se que conduza a uma carga mais baixa que todos os outros atores, por isso, o *Actor Random* servirá como referência para os outros testes. Nos dois gráficos a seguir pode-se ver o desempenho dos computadores à medida que se vão introduzindo mais atores na simulação, um gráfico mostra as medições feitas ao processador e o gráfico a seguir mostra as medições da memória *RAM* usada.

Nos gráficos apresentados de seguida (*Figuras 51 e 52*), são representados as percentagens da carga de processador e memória RAM utilizados em função do número total de atores. O valor total de atores que aparece nos gráficos é o somatório de todos os atores criados em cada um dos *LocalCoordinators*. A distribuição dos atores é mais ao menos de ¼ por cada *LocalCoordinator* no caso em que se faz simulação com 5 computadores e uma proporção de ½ para cada *LocalCoordinator* no caso em que são usados 3 computadores.

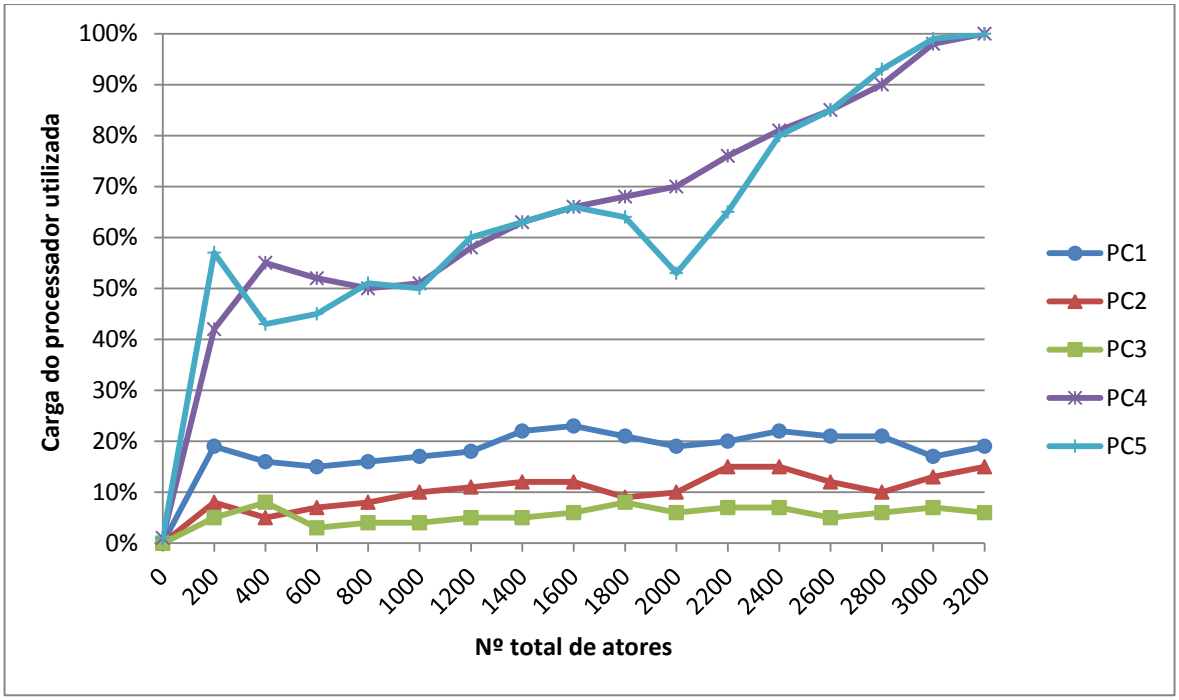


Figura 51 - Gráfico da evolução da utilização do processador com os atores *Random*.

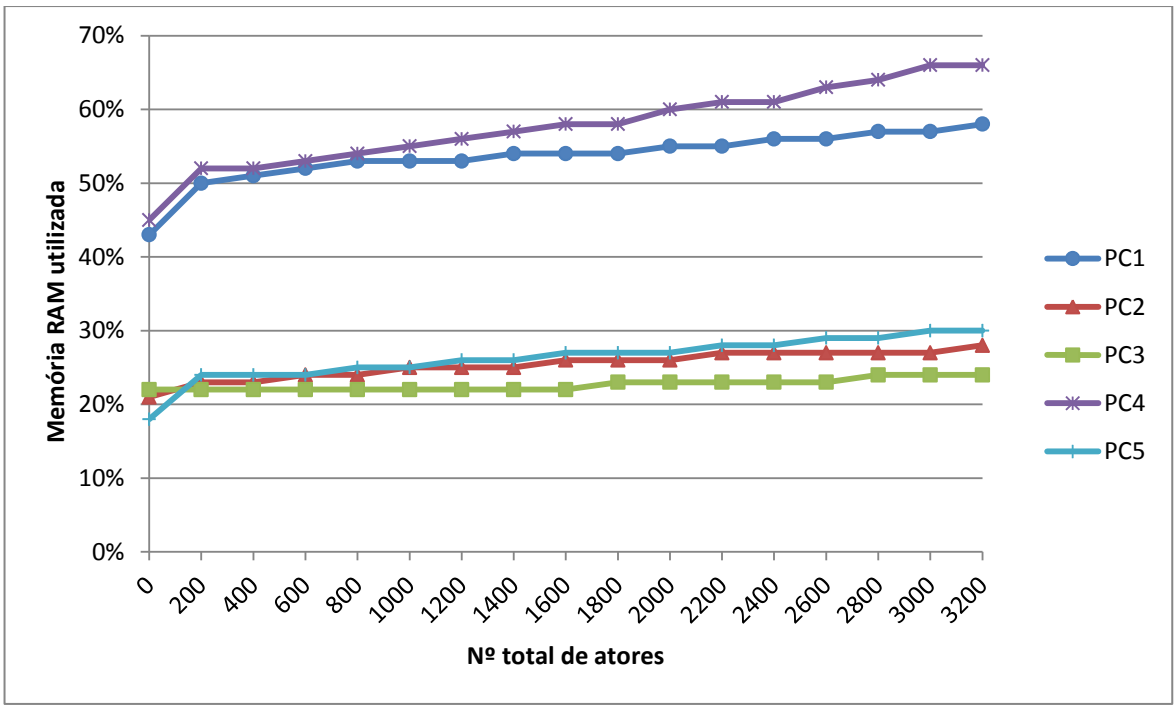


Figura 52 - Gráfico da evolução da utilização da memória *RAM* com os atores *Random*

No segundo teste foram usados os computadores 1 (*GlobalCoordinator*), 2 (*Visualization* e *LocalCoordinator*) e 3 (*LocalCoordintator*) para testar os seus

desempenhos até chegar a um nível considerável de atores. Os computadores 4 e 5 foram deixados de fora dos testes devido a falta de capacidade de atingir grandes quantidades de atores, como se pode ver nos gráficos anteriores, ao fim de 200 atores os computadores 4 e 5 atingiram os 100% da carga do processador.

Foram feitas medições até atingir aproximadamente os 3500 atores por cada *LocalCooordinators*. Foi estabelecido este valor devido a uma limitação descoberta na altura destes testes, em que cada máquina não se conseguia criar mais do que 4000 atores por computador. Ao tentar levar a simulação até ao limite máximo, ocorreu um problema com o simulador, que deixou de criar atores num dado momento. Assumiu-se que seriam limitações do próprio sistema operativo, pois chegava aproximadamente aos 4000 atores e o *LocalCoordinator* deixava de responder aos pedidos do *GlobalCoordinator* de criação de novos atores. Para evitar que os dados obtidos fossem alterados definiu-se um limite de 3500 atores por *LocalCoordinator*.

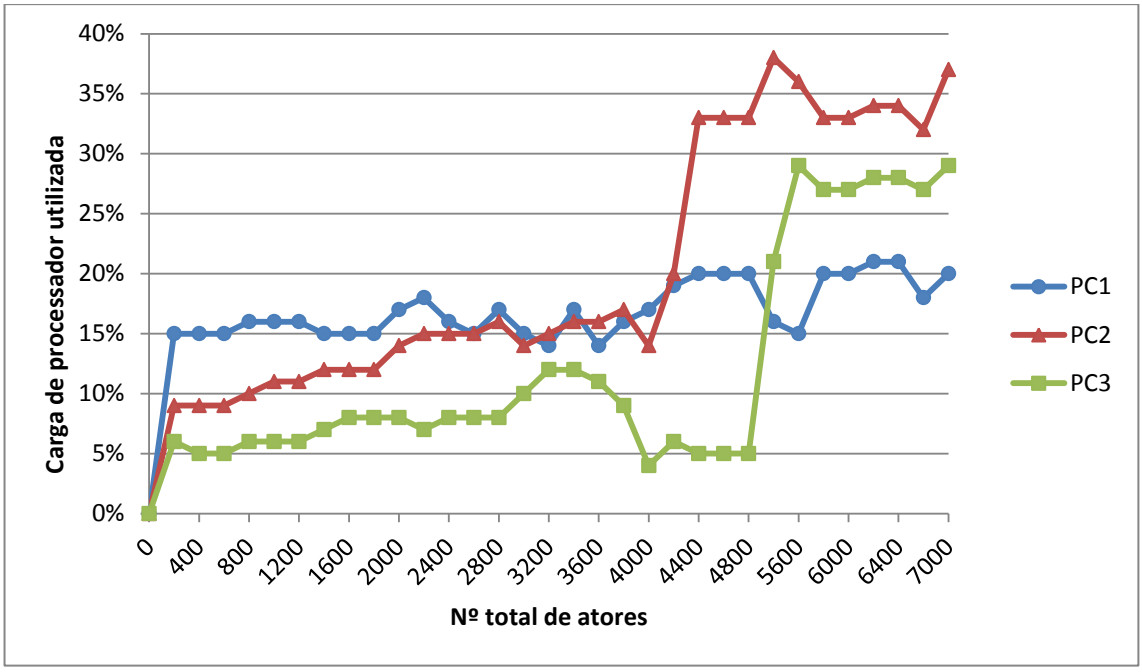


Figura 53 - Gráfico da evolução da utilização do processador com os atores *Random*

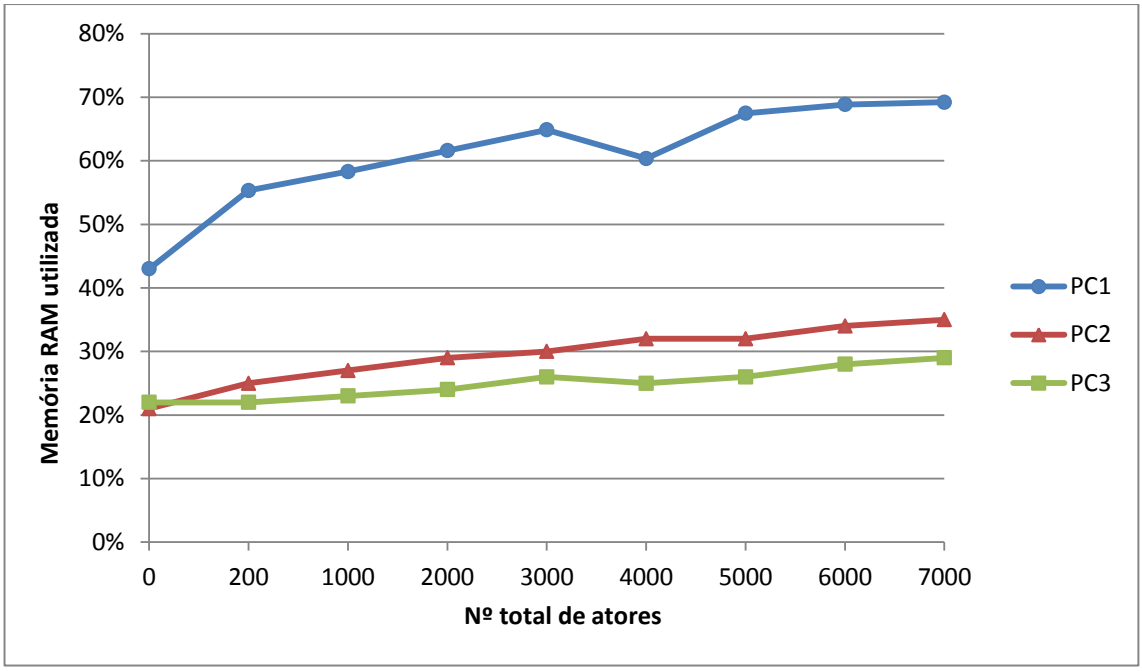


Figura 54 - Gráfico da evolução da utilização da memória *RAM* com os atores *Random*

No primeiro gráfico (*Figura 53*) os valores do PC 1 não sofrem grandes alterações, mantendo-se durante a simulação no intervalo de 15% a 25%, sendo um bom valor, tendo em conta as capacidades do PC1. Os PC 2 e 3 apresentaram em momentos muito próximos um aumento muito grande da carga do processador utilizada (PC 2 e 3 com 4000 e 5400 atores respetivamente), sendo que não era esperado um aumento tão acentuado e que até ao momento da escrita da dissertação não se conseguiu perceber o que poderia estar na origem dessa situação.

O segundo gráfico (*Figura 54*), com os valores de utilização da memória RAM, tem um comportamento esperado, ou seja, todos os computadores apresentam um aumento da memória usada à medida que são criados mais atores sem grandes oscilações. Os valores observados no PC1 são mais elevados do que o dos outros computadores, pois tem muito menos memória RAM. O PC2 tem valores um pouco maiores pois estava a executar um *LocalCoordinator* e o *Visualization*, enquanto o PC3 apenas estava a executar o *LocalCoordinator*. A diferença de valores entre os dois últimos computadores é muito pequena, pelo que se pode verificar que o *Visualization* tem pouco impacto no desempenho do computador.

### *5.2.2 Cenário de teste 2: Actors Tram*

Todos os atores descritos de seguida precisam de usar o mapa para se deslocar e têm que dar atenção aos outros atores que se deslocam à sua volta.

Era esperado que os computadores mais fracos conseguissem suportar algumas dezenas de atores Tram numa simulação. Ao fim de apenas uma dezena de atores os processadores dos computadores 4 e 5 atingiram os 100%. Por não terem capacidade suficiente para simular atores mais complexos, os dois computadores foram removidos do cenário de teste, ficando só os computadores 1, 2 e 3 e, ficando assim, com um *GlobalCoordinator*, um *LocalCoordinator* e um *Visualization*.

Nos dois gráficos a seguir (*Figuras 55 e 56*) estão representados os valores obtidos do processador e da memória RAM nos testes com os atores Tram.

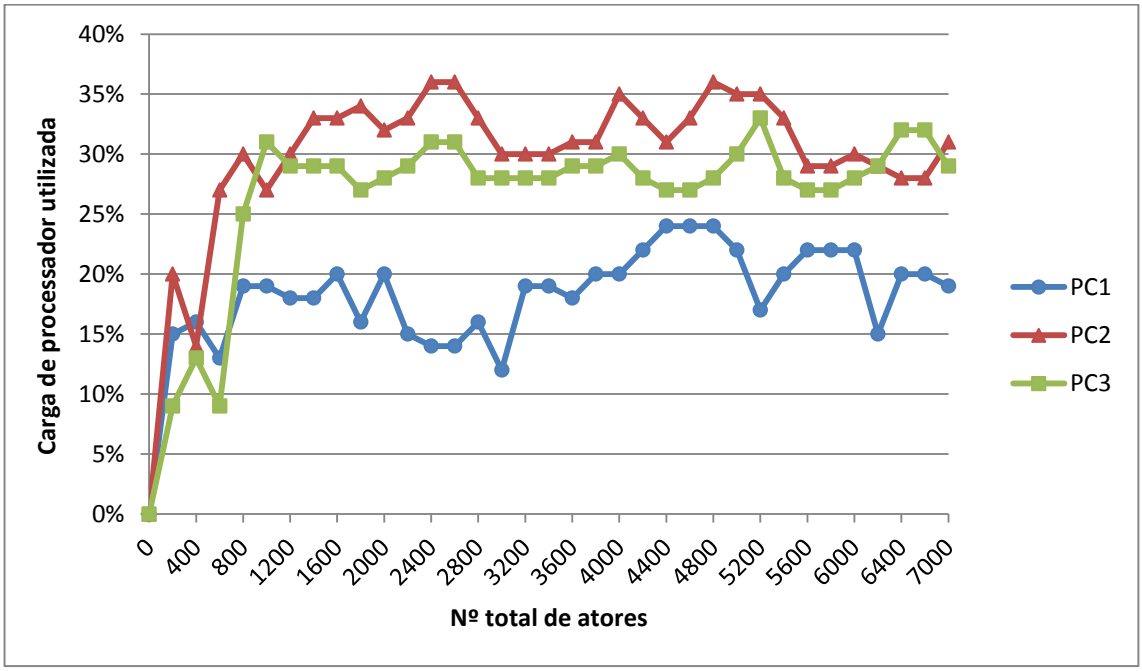


Figura 55 - Gráfico da evolução da utilização do processador com os atores *Tram*

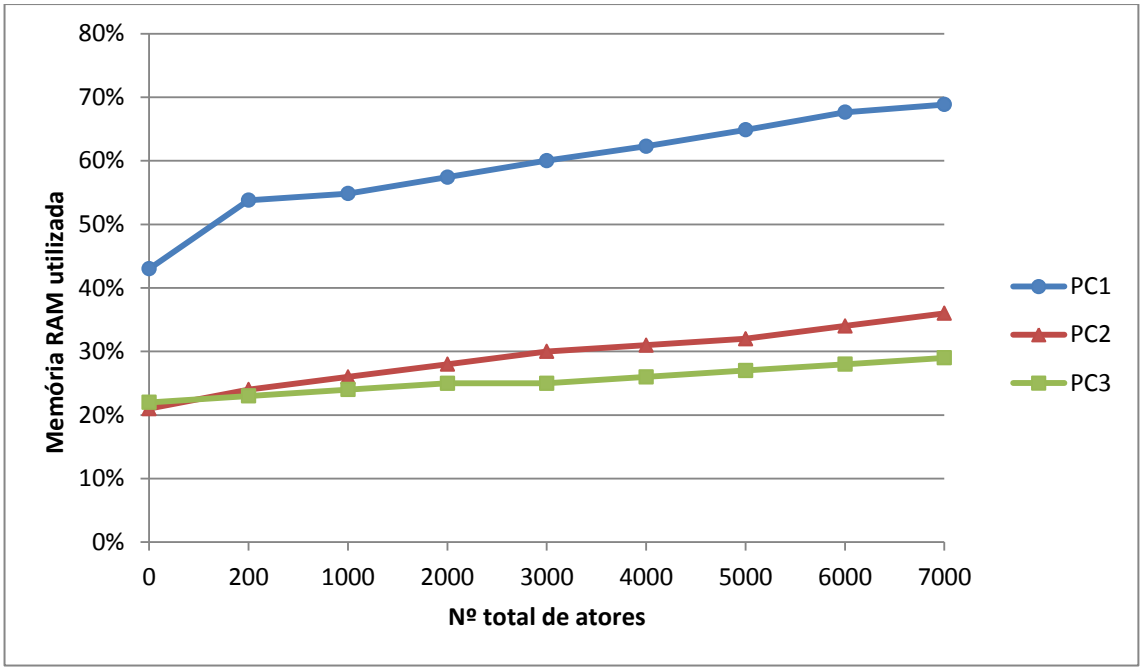


Figura 56 - Gráfico da evolução da utilização da memória *RAM* com os atores *Tram*

No gráfico do processador gasto (*Figura 55)*, tem um aumento rápido inicial, depois estabiliza entre o intervalo 10% e 25% (PC1) e 25% e 40% (PC2 e PC3), tendo um pico máximo de 24% (PC1), 36% (PC2) e 34% (PC3). A estabilidade verificada no gráfico é devida ao facto de os computadores estarem a distribuir a carga pelos vários processadores, tendo sido assim possível, manter os valores mais ao menos uniformes. Um pormenor detetado foi o facto de apenas 3 *cores* dos computadores estavam a ser usados, para fazer a distribuição de carga.

Os valores verificados nesta simulação são superiores aos do *Actor Random*, sendo este um valor esperado, visto que estes atores são mais complexos que os *Random*. O computador 1 apenas atingiu o valor máximo de 24%, um pouco mais alto do que do ator anterior de 21%, mas também tem valores muito próximos dos 20%, não havendo assim nada de muito relevante para reportar.

No segundo gráfico (*Figura 56*) é possível ver uma evolução semelhante ao gráfico da utilização da memória *RAM* do *Actor Random*, ou seja, igual ao *Actor Random*. Não tem grandes alterações, com os valores compreendidos entre 20 e 40% (computador 2 e 3) e, mesmo tendo atores mais complexos, este valor não sofre grandes alterações.

### *5.2.3 Cenário de teste 3: Actors Car*

Neste cenário foi mantido o cenário de teste usado no ator Tram devido à falta de capacidade dos computadores 4 e 5 de processarem grandes quantidades de atores.

Os atores *Car* necessitam de um mapa para se deslocar e têm que dar atenção aos atores vizinhos. O ator Car em relação ao ator Tram tem um comportamento mais complexo, por isso, é esperado que a carga computacional seja superior.

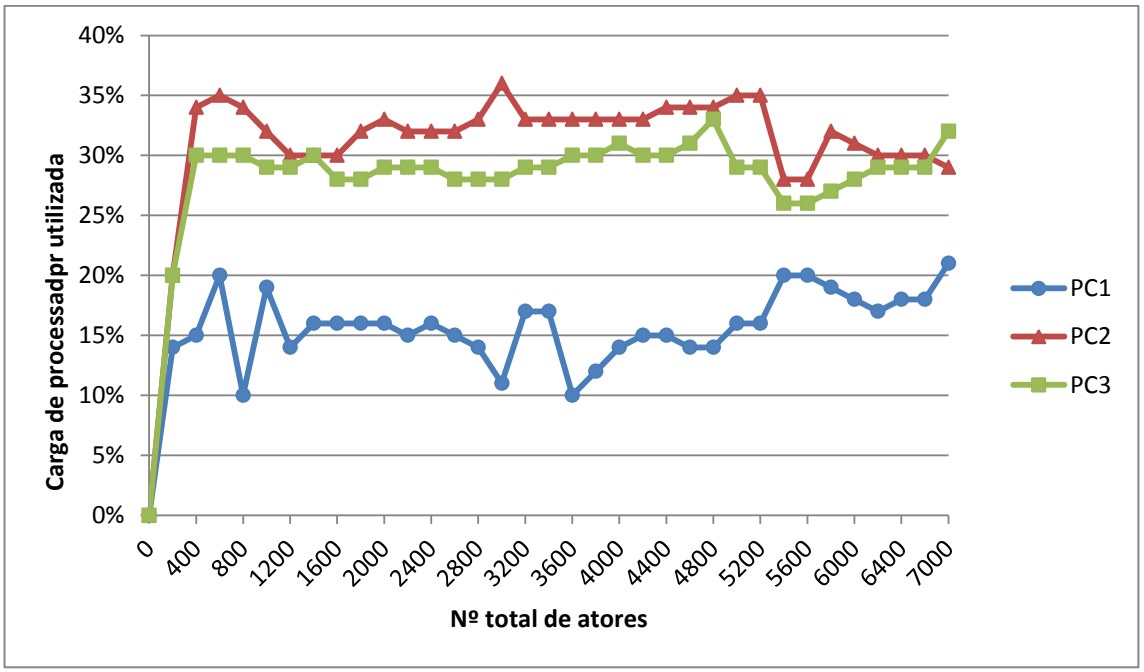


Figura 57 - Gráfico da evolução da utilização do processador com os atores *Car*

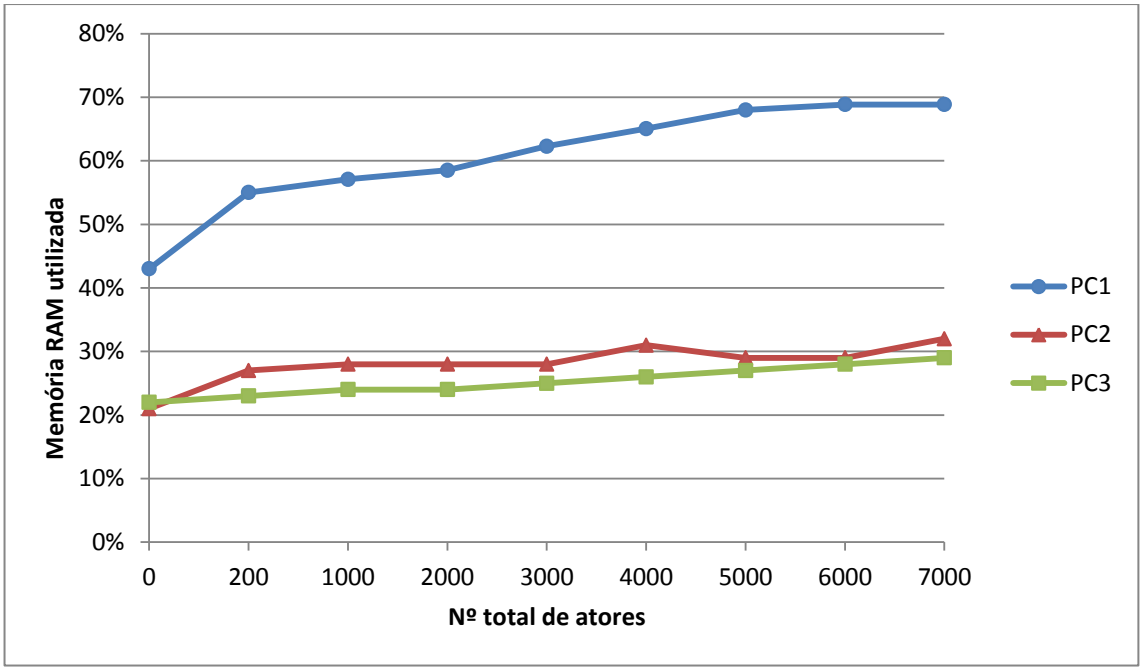


Figura 58 - Gráfico da evolução da utilização da memória *RAM* com os atores *Car*

O primeiro gráfico (*Figura 57*) nota-se que, logo de início tem um grande aumento dos valores de processamento gasto, ou seja, tem um crescimento muito mais acentuado que na situação dos *Actors Tram*. De resto, os três computadores têm um comportamento muito semelhante ao do ator anterior, também atingindo uma estabilidade com os valores muito semelhantes aos verificados anteriormente. Os valores máximos atingidos pelos computadores foram 21% (PC1), 36% (PC2) e 33% (PC3)

No segundo gráfico (*Figura 58*), como explicado no ator anterior, não se verificam grandes variações. Apenas de constatar os valores um pouco mais abaixo que dos atores anteriores e que nos 7000 atores os valores do computador 1 e 2 rondavam os 30%.

### *5.2.4 Cenário de teste 4: Actors Pedestrian*

Os atores Pedestrian têm um comportamento mais simples que os atores Tram e Car, por isso era esperado uma carga computacional inferior.

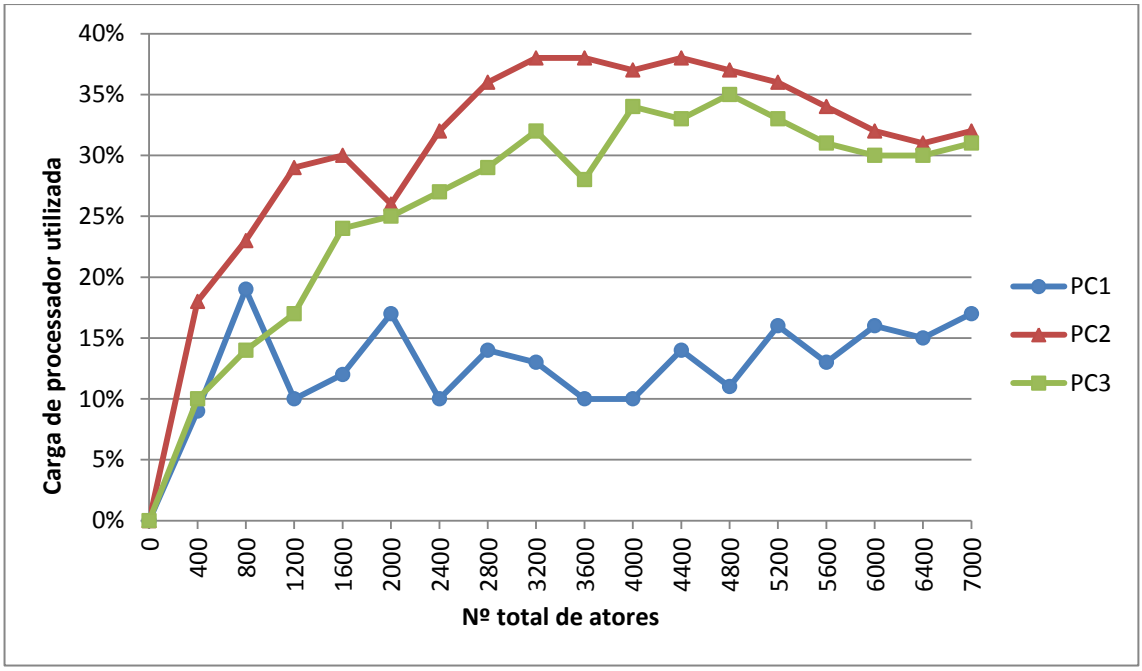


Figura 59 - Gráfico da evolução da utilização do processador com os atores *Pedestrian*

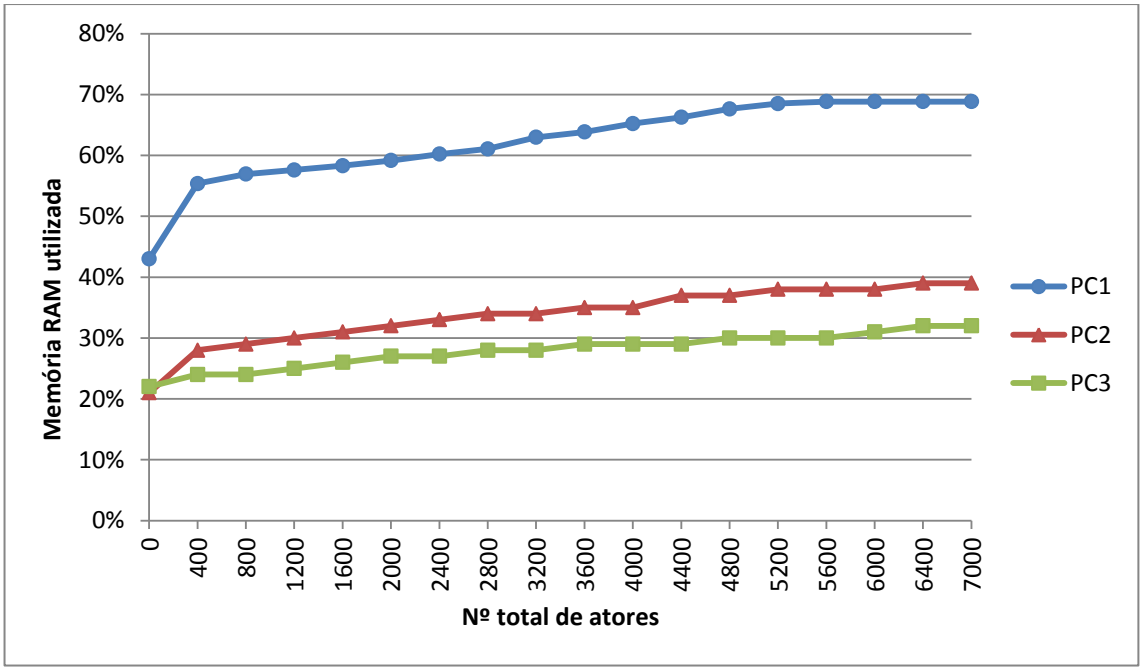


Figura 60 - Gráfico da evolução da utilização da memória *RAM* com os atores *Pedestrian*

Nos dois gráficos (*Figuras 59 e 60*), nota-se um comportamento muito semelhante aos dos atores *Tram* e *Car*. No gráfico do processamento usado (*figura 58*) verifica-se um aumento inicial mais "suave" do que o verificado no *Actor Car*, atingindo depois um valor máximo de 38% (PC2) e 35% (PC3), no entanto, não se verificou

novamente este valor pois o computador começou a usar os vários processadores para distribuir a carga. A partir dos 4400 verifica-se uma descida muito constante mas dentro dos valores alcançados pelos outros Actors com o mesmo número de atores na simulação. O PC1 atingiu uma estabilidade no intervalo de 10% a 20% tendo atingido um máximo de 19%.

### *5.2.5 Cenário de teste 5: Actors Tram, Actors Car e Actors Pedestrian*

Neste cenário de teste será o cenário mais próximo da realidade, pois serão utilizados os atores *Tram*, *Car* e *Pedestrian*. Espera-se que a carga computacional atinja um valor intermédio dos atores anteriores.

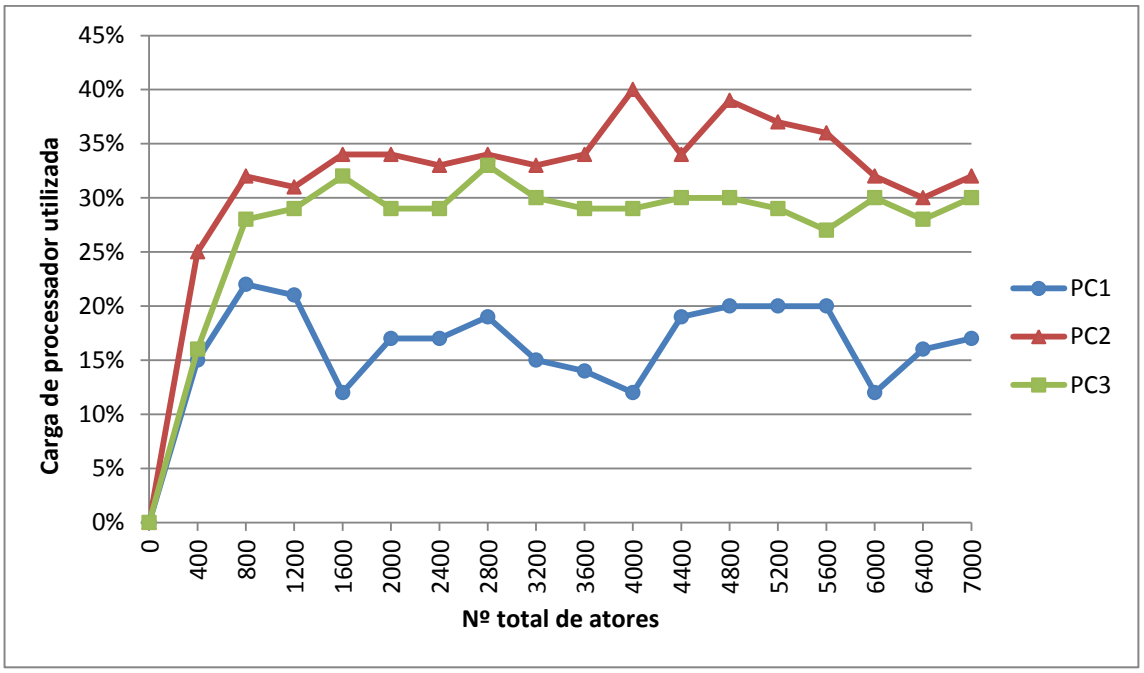


Figura 61 - Gráfico da evolução da utilização do processador com os vários atores

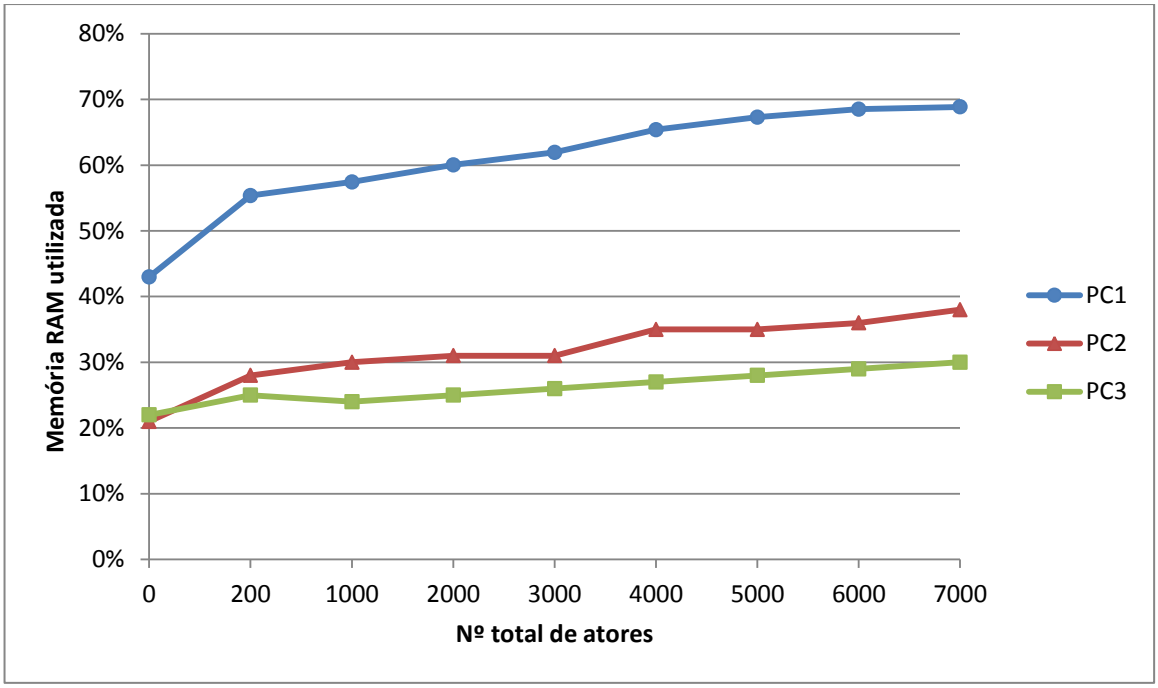


Figura 62 - Gráfico da evolução da utilização da memória *RAM* com os vários atores

Nos dois gráficos (*Figuras 61 e 62*) nota-se um comportamento muito semelhante aos atores *Tram*, *Car* e *Pedestrian*. No gráfico do processador (*Figura 61*) verifica-se um aumento inicial muito semelhante ao verificado no *Actor Tram*, atingindo rapidamente uma estabilidade entre intervalo 25% e 35% e, de seguida, um valor máximo de 40% (PC2) e 33% (PC3). No caso do PC1, atinge uma estabilidade entre o intervalo de 10% a 20%, atingindo um valor máximo de 22%.

## 5.3 Análises de resultados

Após uma análise aos resultados obtidos consegue-se tirar algumas conclusões em relação à carga computacional que cada ator tem. A conclusão que mais se destaca é a diferença de carga computacional do ator *Random* para todos os outros. Quando se testou os atores *Random* com os computadores mais fracos ainda se conseguiu atingir um valor consideravel de atores, quando se tentou simular os outros atores, foi impossível fazer a simulação. Era esperado que os computadores 4 e 5 conseguissem simular pelo menos algumas dezenas de atores. O facto de ao fim de 10 atores os

processadores estarem a 100%, mostra claramente que os atores mais complexos necessitam de mais memória Ram para poderem ser executados.

Quando se simulou o ator *Random* com os computadores melhores, os processadores dos *LocalCoordinator* (computador 2 e 3) até aos 2000 atores não chegaram aos 20%, nos testes efetuados com os outros, logo de início, ultrapassou-se a barreira dos 20% e apenas estabilizou no intervalo entre 30% e 40%.

Como se pode ver, perante os resultados obtidos em todos os tipos de atores, o *GlobalCoordinator* (computador 1) manteve uma carga constante sobre o computador, entre 15% e 25%, e a carga nos *LocalCoordinator* foi evoluindo, à medida que se ia aumentado o número de atores associados a cada um deles. Estes dois comportamentos já eram esperados, de referir que o *GlobalCoordinator* se encontrava num computador mais modesto, sendo por isso um excelente resultado. Já os valores atingidos pelos *LocalCoordinator* encontram-se um pouco acima do esperado. De forma a ser possível se poder simular milhares de atores em cada computador é necessário que cada ator gaste o mínimo possível de recursos. Nos atores existem dois métodos críticos que poderão originar gastos em excesso se forem mal implementados, os métodos são o *findNextDestination()* e o *getNeighbours()*. No caso do primeiro método, o ator, sempre que precisa de um novo destino, tem de procurar por todo o mapa. Se o mapa for demasiado grande e forem demasiados atores à procura de um ponto, pode gerar muita carga extra sobre o computador. Já o segundo método, torna-se cada vez mais crítico com o aumento dos atores na simulação, pois, a cada atualização de um ator, tem de procurar em todos os atores se algum é seu vizinho, o que faz com que este método não seja escalável.

De forma a perceber se os dois métodos seriam muito críticos para o desenrolar da simulação, foi feita uma medição ao tempo que cada método demorava a ser executado durante uma simulação. Com mais ao menos 500 atores o método *findNexDestination(),* em situações mais críticas, poderia atingir até 80 milisegundos. No caso do *getNeighbours(),* à medida que os atores iriam aumentando, poderia atingir os 270 milisegundos. Claramente, os dois métodos têm grande influência no

processamento necessário do simulador. Portanto, estes métodos precisam de ser melhorados.

Era esperado que diversos atores tivessem uma carga mais ao menos semelhante entre eles, com exceção do ator *Random*, mas com algumas oscilações de carga devido a especificidade de cada ator. À partida, quanto mais complexos maior a carga sobre o computador. Como se pode ver nos resultados obtidos, inicialmente o valor da carga do processador aumenta, depois mantem-se constante num intervalo entre 25% e 40%. As oscilações mais acentuadas nos computadores 2 e 3 devem-se ao facto dos computadores serem *multi-core* e a carga era distribuída pelos outros *cores*.

Em relação aos resultados da memória *RAM* usada, os valores são bastantes satisfatórios. No computador 1 os valores foram um pouco elevados mas deve-se ao facto do computador ter pouca memória RAM. Já os computadores 2 e 3 atingiram um valor máximo de 39% com 3500 atores *Pedestrian*. Os valores foram crescendo a medida que eram criados mais atores, não havendo alterações bruscas, podendo-se prever que, com mais atores num computador, a carga da memória RAM iria manter o mesmo comportamento, possibilitante a criação de um grande número de atores.

# 6 Conclusão e Trabalho Futuro

Neste capítulo são apresentadas as conclusões gerais da dissertação. Inicialmente é feita uma reflexão do trabalho efectuado e resultados obtidos. Por fim são apresentadas algumas considerações e recomendações para trabalho futuro.

## 6.1 Conclusão

Com o aumento substancial dos dispositivos móveis, pretendeu-se com o desenvolvimento do simulador BartUM que este fosse uma ferramenta importante para os estudos sobre a mobilidade dos dispositivos móveis em ambientes urbanos.

Para ajudar no desenvolvimento do simulador, foi feito um estudo sobre simuladores semelhantes ao que se pretendia implementar, modelos de mobilidade mais relevantes, níveis de simulação e tipos de simulação existentes. O conhecimento destas tecnologias permitiu um melhor entrosamento na área de simulação.

O trabalho apresentado está a funcionar e está implementado da forma que se pretendia, os geradores criam corretamente todos os tipos de atores, os atores têm conhecimento dos outros atores do próprio LocalCoordinator e dos outros LocalCoordinators da rede. Os atores *Tram*, *Car* e *Pedestrian* foram pensados de forma a serem uma representação próxima da realidade, e o ator Random foi pensado para simular movimentos erráticos, bem como, todos os comportamentos pensados para cada um deles foram implementados com sucesso. Através da mesma base do *Actor Generic*, conseguiu-se desenvolver atores diferentes e com procedimentos diferentes para as diversas situações com que se depararam.

Para realizar as simulações com o máximo potencial dos computadores, é importante ter os computadores apenas a executar o simulador e não estarem a ser usados recursos extras no computador, criando assim uma maior estabilidade para a simulação. Para perceber a influência que o simulador tinha sobre os computadores, foi

necessário ter os computadores estáveis. Através dos testes apresentados pode-se verificar que o esforço computacional exigido pelos diversos atores foi um bom indicador para se realizar simulações de grandes dimensões. O *GlobalCoordinator* nos testes, apesar de estar colocado num computador mais modesto, teve um bom desempenho, aguentando com um número considerável de atores.

Não foi detetado nenhum problema de estabilidade por parte do simulador, apesar de quando se tentou levar a simulação até ao máximo de atores possíveis, não se conseguiu criar mais do que 4000 atores, sendo possivelmente um problema do sistema operativo. Com os indicadores retirados dos testes efetuados é expectável que o simulador consiga criar uma grande quantidade de atores, uma vez ultrapassado este problema.

## 6.2 Trabalhos Futuros

Para trabalhos futuros será prioritário conseguir melhor o desempenho dos atores ao nível de gasto de recursos, principalmente nos métodos já referidos. Será importante criar novos tipos de atores como autocarros e bicicletas, implementar um sistema de sinalização e criar métodos de paragens em locais específicos para os transportes públicos, por exemplo. Será necessário implementar o sistema de comunicação entre atores, pois é um dos objetivos principais do simulador, que irá ajudar nos estudos das redes móveis num ambiente urbano.

Como as simulações vão atingir quantidades enormes de atores, é importante conseguir criar soluções para gerir a carga gasta por multidões de pedestres ou grandes filas de trânsito, ou seja, o simulador perceber que os vários atores terão comportamentos semelhantes e poder assim gerir recursos computacionais, evitando que todos os atores estejam a calcular constantemente a sua posição, poupando assim recursos do computador.

O tempo de simulação decorre em tempo-real, para já as simulações não podem decorrer com tempo de simulação, mas é uma capacidade que o simulador precisa de ter de forma a realizar simulações de horas em apenas poucos minutos e vice-versa. Será também importante fazer medições ao tráfego na rede local em simulações futuras, para se perceber a quantidade de informação que é trocada entre os vários intervenientes da simulação. Um último melhoramento necessário é a função de distribuição de carga, que está a fazer a distribuição pelo número de atores em cada *LocalCoordinator*, o que pode gerar carga a mais num *LocalCoordinator* que esteja num computador mais fraco.

# 7 Referências


Barceló, Jaume. (2010) *Fundamentals of Traffic Simulation*. Department of Statistics & Operations, Research. Universitat Politècnica de Catalunya, Barcelona. Spain.

Bezerra, Rafael Lopes. (Março 2009). *Análise da conectividade em redes móveis utilizando dados obtidos da mobilidade humana*. Rio de Janeiro.

Boldrini, Chiara; Conti, Marco; Passarella, Andrea. (2009) *The Sociable Traveller: Human Travelling Patterns in Social-Based Mobility*. IIT-CNR, Via G.Moruzzi 1, Pisa, Italy.

Bonnmotion. (Agosto 2012). Acedido a Outubro de 2012, em http://net.cs.uni-bonn.de/start/

Campos, Carlos Alberto V.; Moraes, Luís Felipe M. (2007). *Uma Proposta de Caracterização da Mobilidade de Usuários Sem Fio Através de Medição Real*. Laboratório de Redes de Alta Velocidade- RAVEL, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ. Rio de Janeiro.

Camp, Tracy; Boleng, Jeff; Davies, Vanessa. (2002) *A Survey of Mobility Models for Ad Hoc Network Research*. Wireless Communication & Mobile Computing. Colorado School of Mines, Golden. Colorado Setembro.

Gnuplot. (Março 2012). Acedido a Outubro 2012, em http://www.gnuplot.info/

Hsu, Wei-jen; Merchant, Kashyap; Shu, Haw-wei; Hsu, Chih-hsin; Helmy, Ahmed. (2005). *Weighted Waypoint Mobility Model and its Impact on Ad Hoc Networks*. Computer Science Department, Electrical Engineering Department, University of Southern California. California.

*Informatik 4:BonnMotion*. (2012, Outubro). (University of Bonn) Acedido a Outubro 2012, em http://net.cs.uni-bonn.de/wg/cs/applications/bonnmotion/

Keränen, Ari, Jörg Ott, and Teemu Kärkkäinen. (2009). *The ONE Simulator for DTN Protocol Evaluation, Department of Communications and Networking*. Article, Department of Communications and Networking, Helsinki University of Technology. Helsinki.

Mello, Braulio. (Outubro 2001). *Modelagem e Simulação de Sistemas*. [Versão electrónica]. URI-Universidade Regional Integrada do Alto Uruguai e das Missões Departamento de Engenharias e Ciência da ComputaçãoCiência da Computação - Sistemas de Informação. Acedido a Outubro 2012, em http://pt.scribd.com/doc/51072551/10/Classificacao-dos-modelos-de-simulacao.

Musolesi, Mirco, e Cecilia Mascolo. (2008). *Chapter 1 Mobility Models for Systems Evaluation A Survey*. Dartmouth College; University of Cambridge, 28.

Nain, Philippe; Towsley, Don; Liu, Benyuan; Liu, Zhen. (2005). *Properties of Random Direction Models*.University of Massachusetts, Amherst, MA 01002. USA.

Ni, Daiheng. (2006). *A Framework for new Generation Transportation Simulation*. University of Massachusetts, Amherst, MA.

Nunes, Bruno Rios Patriarca. (2009). *Especificação e Protocolo para a Gestão da Filiação ao Grupo em Redes Móveis Ad Hoc*. Departamento de Ciência da Computação, Instituto de Matemática, Universidade Federal da Bahia. Salvador.

Olstam, Johan Janson. (2005). *A model for simulation and generation of surrounding vehicles in driving simulators*. Department of Science and Technology, Linköpings Universitet, SE-601 74 Norrköping, Sweden. Norrköping.

*OpenJUMP GIS*. (2011). Acedido a Outubro 2012, em OpenJUMP GIS: http://www.openjump.org/

Open Street Map. Acedido a Outubro 2012, em http://www.openstreetmap.org/

Ribeiro, Andrea, e Rute C. Sofia. (2011). *A Survey of Mobility Models on Wireless Networks*. SITI Technical, Lusífona University. Lisboa.

Silva, Francisco Manuel A. R. (Outubro 2011) *Simulador de Movimento oara Ambientes Urbanos*. Universidade do Minho. Azurém, Guimarães.

*SUMO - Simulation of Urban Mobility*. (2012). Acedido a Outubro 2012, em http://sumo.sourceforge.net/

*The ONE*. (n.a.). Acedido a Outubro 2012, em The ONE: http://www.netlab.tkk.fi/tutkimus/dtn/theone/

Vasconcelos, António Luís P. (Fevereiro, 2004). *Modelos de Atribuição/Simulação de Tráfego: O Impacto na Qualidade dos Resultados de Erros no Processo de Modelação*. Faculdade de Ciências e Tecnologia da Universidade de Coimbra. Coimbra.